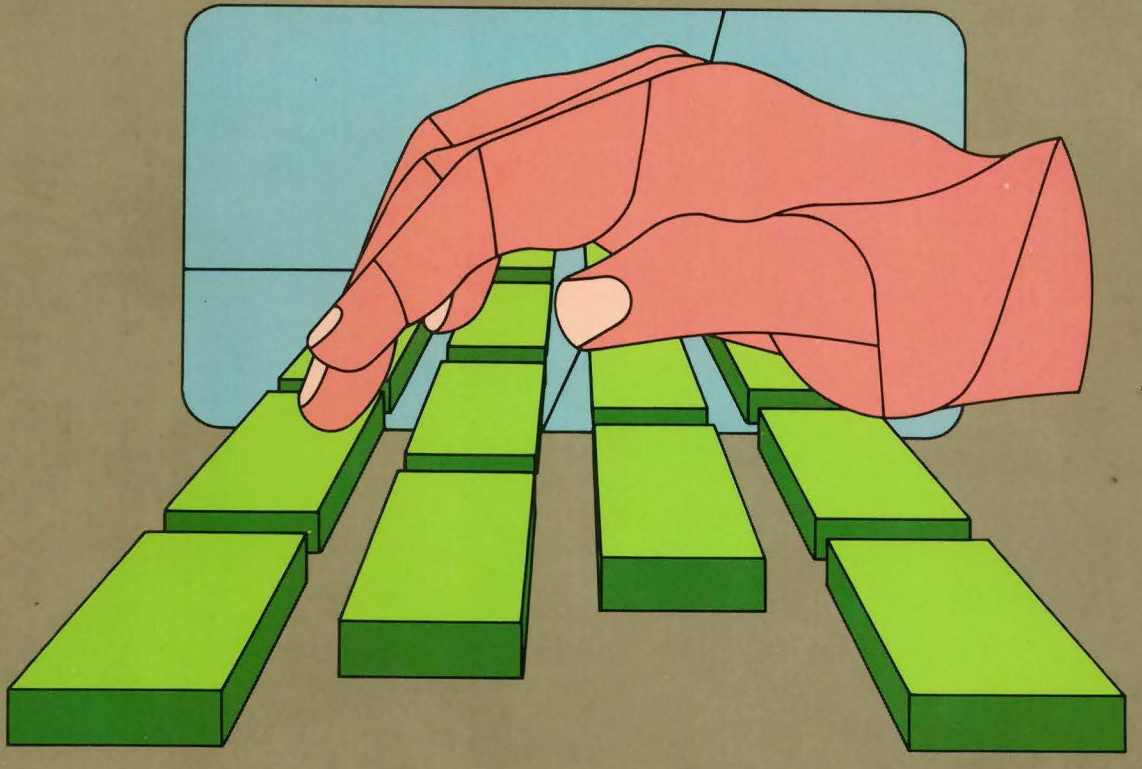
T/S 1500 T/S 1000 ZX-81

Basics of Timex Sinclair 1500/1000 BASIC

Allen Wolach

A wonderful introduction to the world of BASIC!



BookWare™ *For the Timex Sinclair 1500, Timex Sinclair 1000 and ZX-81 Computers*

Basics of Timex Sinclair 1500/1000 BASIC
Allen Wolach
BookWare
T/S 1500
T/S 1000
ZX-81
R0347-6

Basics of Timex® Sinclair 1500/1000™ BASIC

Allen H. Wolach
Illinois Institute of Technology

A Reston Computer Group Book
Reston Publishing Company, Inc.
A Prentice-Hall Company
Reston, Virginia

Previously published as *Basics of Timex® Sinclair 1000™, ZX81 BASIC.*

ISBN 0-8359-0347-8

This book, and the programs in it, can be used with the Timex Sinclair 1500 and 1000 computers, and with the Sinclair ZX81. References in the book to the ZX81 apply equally to the T/S1500 and T/S1000.

RESTON PUBLISHING COMPANY, INC.
A Prentice-Hall Company
Reston, Virginia

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

Contents

Preface

This book is intended as an introduction to programming in BASIC. The version of BASIC used throughout the book is the BASIC found in Sinclair ZX81 microcomputers and Timex-Sinclair 1000 and 1500 microcomputers. The book can also be used with Sinclair ZX80 microcomputers and Micro Ace microcomputers that are updated with the Sinclair 8K ROM. Updated ZX80 and Micro Ace microcomputers lack a few of the features found in the ZX81 and Timex-Sinclair 1000 and 1500 microcomputers.

The present book is different from the manual that comes with a Timex-Sinclair computer. A manual must highlight each of the features of a microcomputer. Sometimes these features are emphasized at the expense of explaining the BASIC programming language. Programming is emphasized in this book. For example, the LET statement is explained in detail. The reader will learn how to use the LET statement to initialize variables,calculate results from a formula, sum numbers, and so forth.

Many recent books, including the ZX81 manual, are what I call "learn by doing" books. The microcomputer user enters a program sequence that is shown in the book. Then the text explains the output that the microcomputer produces. This kind of book has two major problems. If the book is read when the microcomputer is not available, the reader is not sure of the output that would be produced by the microcomputer. Sometimes a microcomputer user incorrectly enters a program sequence and draws the wrong conclusion about the program sequence. A "learn by doing" book does not always show the correct interpretation of a program sequence. A reader is supposed to infer the correct interpretation from what

appears on the television screen. Sometimes a microcomputer user is supposed to learn about BASIC by performing exercises that are at the end of a chapter. If the user cannot perform an exercise, he cannot learn what the exercise is intended to teach. The major advantage of a "learn by doing" book is that the user is forced to interact with the microcomputer. This interaction helps the user retain what he learns.

The present book is not a "learn by doing" book. The book can be read when the microcomputer is not available. Since the book is complete, the reader cannot easily draw an incorrect conclusion about a program sequence. The book is written so that the reader can try program sequences in the microcomputer. The reader has the option of interacting with the microcomputer or reading the text when the microcomputer is not available. Since most of the available books on Sinclair BASIC are "learn by doing" books, the present book provides a valuable alternative for microcomputer users who are uncomfortable with the "learn by doing" approach.

Chapter 1

The Keyboard

Command versus Program Mode

Sinclair 8K ROM microcomputers can be used in two modes, COMMAND and PROGRAM. The COMMAND mode is used when the computer is operated much like a calculator. COMMANDs are executed as soon as they are entered in the microcomputer. A COMMAND must start with a keyword. The PROGRAM mode is used to execute a series of program statements that are stored in the microcomputer. Each program statement must start with a number. This number can be any integer between 1 and 9999.

Cursor and Key Strokes

When the Sinclair microcomputer is plugged in, a cursor appears at the lower left hand corner of the television screen. This cursor is a small black square with a white K inside. The K stands for keyword. This cursor is sometimes referred to as the inverse K cursor because the K is white as opposed to black. The first key that is pressed must be a keyword if a COMMAND is being entered, or a number if a PROGRAM statement is being entered. Each key on the keyboard can have as many as five functions. However, the first key stroke after the microcomputer is turned on can only be used to enter a digit of a number, or a keyword.

Keyboard

Figure 1-1 shows the keyboard of a Sinclair microcomputer. Figure 1-2 shows what will be entered on the television screen if a given key is pressd when the K cursor first occurs on the screen.

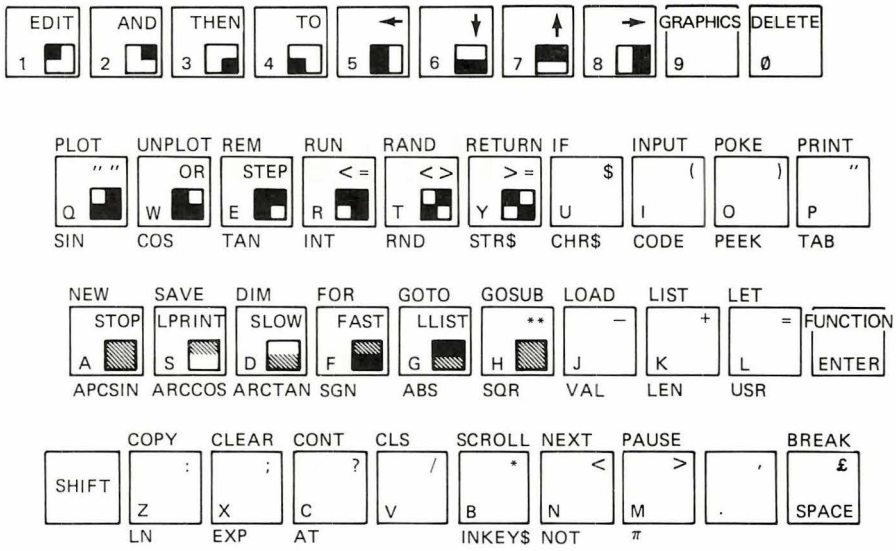


Figure 1-1. Keyboard of a Sinclair 8K ROM microcomputer.

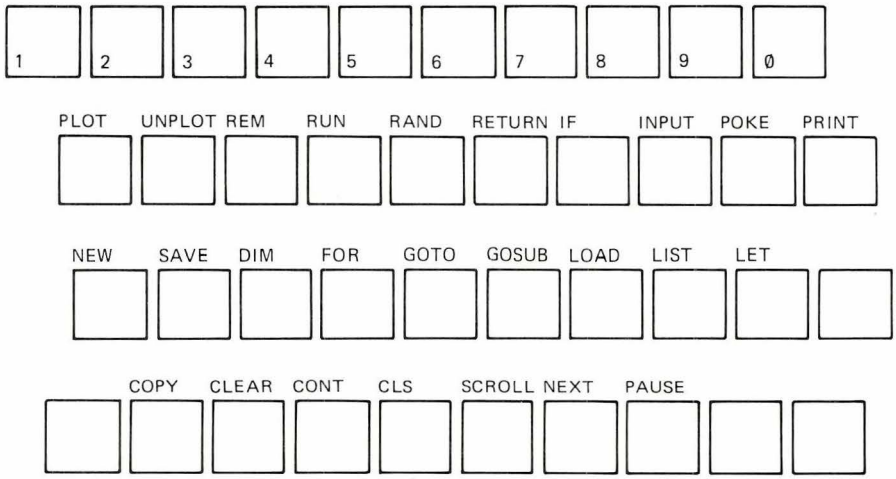


Figure 1-2. The first key stroke can be used to produce any of the above numbers or keywords.

Statement Numbers or Commands

The first key stroke can only enter a number 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 or one of the keywords that is shown above those keys that can produce keywords. If a keyword is entered, a COMMAND is started. If a digit is entered, a PROGRAM statement is started. If a user starts a PROGRAM statement, he may press as many as three more digits. Remember that a PROGRAM statement number can be any integer from 1 to 9999. Thus, if a user enters 2, followed by 5, and then presses one of the keyword keys in Figure 1-2, the microcomputer will enter the number 25 followed by one of the keywords shown in Figure 1-2.

Note that the BREAK on the last key in the last row of the keyboard is not a keyword. Later we shall see that BREAK can be used to terminate a program that is not functioning properly. If the BREAK key is pressed as a first key stroke, a space will be entered. That is, the cursor will be moved one place to the right. Leading spaces will be ignored by the computer. If the ninth key in the fourth row is pressed as a first key stroke, a period (.) will be entered on the screen. Since the period is not an integer or keyword, it should not be entered as a first key stroke.

Key Strokes After the Keyword

Once a keyword is entered, pressing a key that can enter a keyword will cause the key to enter something other than a keyword. After a keyword is entered the cursor changes to a black square with a white L inside. The change in the cursor indicates that letters and numbers as opposed to numbers and keywords can now be entered from the keyboard. Since the K cursor and L cursor are white letters in a black square, they are sometimes referred to as an inverse K or an inverse L. Figure 1-3 shows the letters and numbers that can be entered subsequent to entering the keyword.

The first key stroke after a keyword is entered can also be used to enter a period (row four, second to last key) and a space (row four, last key). The period and space are not letters or numbers. However, entering a period or space following a keyword does not necessarily produce an illegal entry.

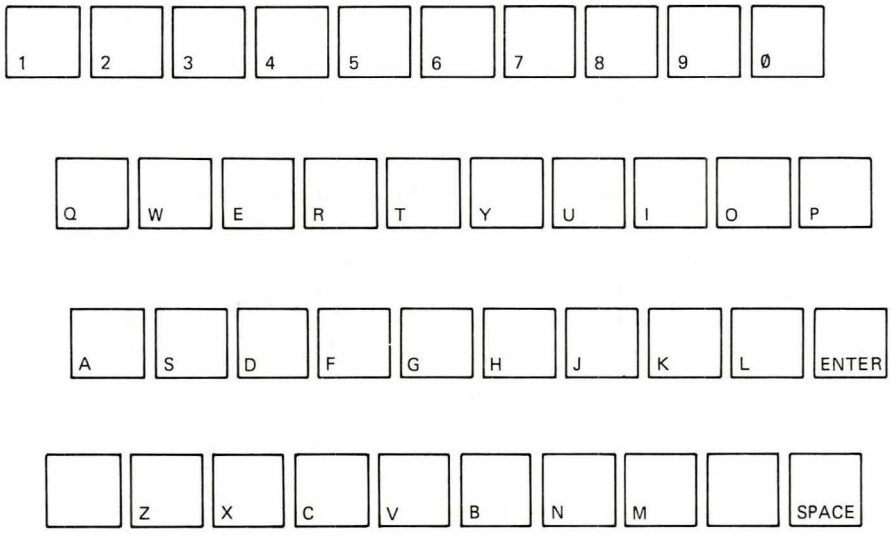


Figure 1-3. Letters and numbers.

The ENTER Key

All COMMANDS and PROGRAM statements are terminated by pressing the key that has ENTER printed on the bottom half (Figure 1-3). If the COMMAND or PROGRAM statement is syntactically correct (follows the rules of the BASIC programming language), the statement is entered in the list of statements at the top of the television screen. A new COMMAND or PROGRAM statement can be entered after the ENTER key is depressed. The first key stroke after ENTER is pressed must again be a digit starting a statement number or a keyword starting with a COMMAND.

Entering a Command

Suppose that the microcomputer has just been turned on. The inverse K cursor appears on the television screen. Next the user presses the key that has the word PRINT shown above the key. We know that a COMMAND is being entered because no digits were entered before the keyword. The word PRINT will appear at the lower left hand side of the television screen after the PRINT key is pressed. Suppose that the user presses the number 5 after the PRINT key is pressed. A 5 will appear next to the PRINT at the

lower left of the screen. Then the user presses the ENTER key to terminate the COMMAND. After a brief pause a 5 will appear at the upper left of the television screen. The user COMMANDed the computer to PRINT the number 5 at the top left of the screen when he entered the sequence

PRINT 5 ENTER

When the COMMAND is executed a 0/0 will appear at the bottom left of the screen. The first 0 indicates that the COMMAND was completed successfully. The second 0 indicates that a COMMAND as opposed to a PROGRAM statement was entered. The 0/0 at the lower left of the screen can now be interpreted as an inverse K cursor. That is, the first key stroke after a 0/0 occurs on the screen will have the same effect as the key stroke to an inverse K cursor. Note that the 0/0 report at the left of the screen has slashes through the 0s. The slashes indicate the number 0 as opposed to the letter O. Always remember that the computer cannot interpret the letter O as a 0 or the 0 as a letter O. The 0 is the last key in the first (top) row of keys and the O is the second to last key in the second row of keys.

Conventions for Showing Key Strokes

Note that the PRINT and ENTER are underlined to indicate that a single key stroke produces the keyword PRINT and a single key stroke is used to ENTER the COMMAND. Since all COMMANDS and PROGRAM statements terminate with a depression of the ENTER key, the ENTER that occurs at the end of each COMMAND or PROGRAM statement will not be shown in the remainder of this book. That is

PRINT 5 ENTER

will be shown as

PRINT 5

Keyboard After Keyword is Entered

We have seen that the key stroke that occurs after a keyword is entered will enter the letter that is at the lower inside of each key.

Shift Key

Sinclair microcomputers have a shift key with SHIFT printed in red. Pressing the SHIFT key and holding it down while another key is pressed will enter one of the red symbols (inside upper right hand corner of each key) on the keyboard. Figure 1-4 shows the symbols that can be entered after the SHIFT key is pressed.

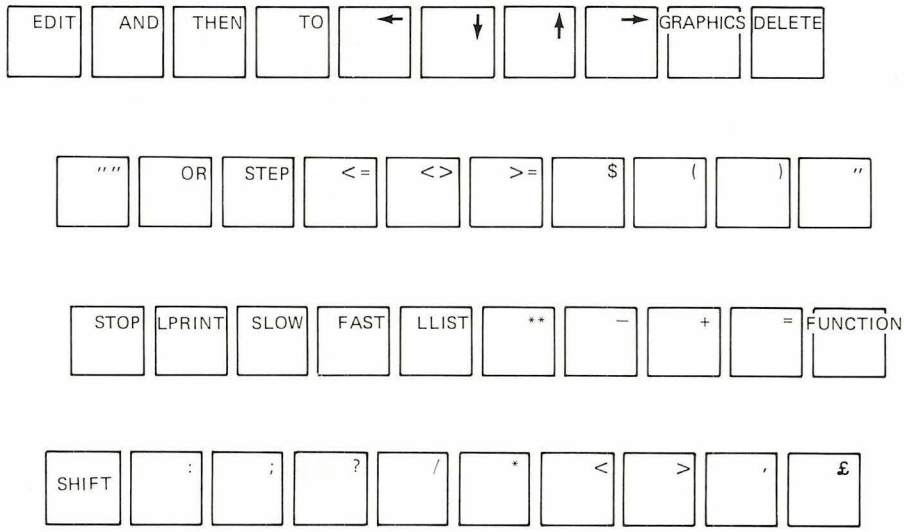


Figure 1-4. Symbols that can be entered after the SHIFT key is pressed.

Note that STOP, LPRINT, SLOW, FAST, and LLIST are keywords. These keywords are obtained by pressing the shift key prior to pressing the key with the appropriate keyword (Figure 1-4). The EDIT, ←, ↓, →, →, and DELETE keys in Figure 1-4 are used to correct or change a program. These keys do not necessarily produce a change on the screen. The EDIT, ←, ↓, →, and → keys will be discussed in detail in a subsequent chapter. The AND, THEN, TO, OR, and STEP keys are used as parts of PROGRAM statements or COMMANDs. They occur after the keyword in the statement or COMMAND. These keys will be discussed in conjunction with the keywords that precede them. All of the other entries that occur after pressing the SHIFT key will be discussed in subsequent sections of this book.

Adding Two Numbers

Suppose that the microcomputer has just been plugged in. Entering the sequence

```
PRINT 5+4
```

will cause the microcomputer to print 9 in the upper left hand corner of the screen. The microcomputer was COMMANDed to PRINT the sum of 5 plus 4. Remember that PRINT is a keyword, numbers such as 5 and 4 are entered by pressing the key that shows the appropriate number. The + is entered by pressing the + key while the SHIFT key is in the pressed position. The line is terminated by pressing the ENTER key.

Errors

Before we discuss the remainder of the keyboard, let us discuss how to correct errors. By now the reader must have discovered that the microcomputer can be cleared by unplugging the power supply and then reinserting the plug. This is not a particularly useful method of clearing the microcomputer. Not only does it require effort, it also clears everything that was entered in the microcomputer.

Sinclair microcomputers have two keys that can be used to move the cursor to the left or right. One key is labeled ← (a shifted 5) and the other key is → (a shifted 8). These keys can be used to move the cursor anywhere in the line that is entered on the bottom of the television screen. A depression of the SHIFT key in conjunction with a depression of → moves the cursor one position to the right. A depression of the SHIFT key in conjunction with a depression of ← moves the cursor one position to the left. If the DELETE key is pressed (do not forget to press the SHIFT key before pressing the DELETE key), the character, number, symbol, or keyword that is to the left of the cursor will be deleted. The entire line at the bottom of the screen can be deleted by moving the cursor to the right of the line with successive depressions of the → key. The successive depressions of the DELETE key will delete the entries on the line.

Since a keyword such as PRINT takes only one key depression to enter, one depression of the DELETE key will delete the entire

keyword. Do not be afraid to press the DELETE key too many times. Extra depressions of the → or ← keys will never move the cursor beyond the beginning or end of the line at the bottom of the television screen.

A character (number, letter, keyword, etc.) can be inserted in a line. Move the cursor to the position where the entry is to be made and then make the entry. If the wrong keyword is in a line, move the cursor to the position that is directly to the right of the keyword, delete the keyword, and then enter the correct keyword. Once the line at the bottom of the screen is corrected, the ENTER key can be pressed. The cursor does not have to be moved to the right of the line before ENTER is pressed.

Checking Syntax

The method for deleting a line that is described above can be very useful. The microcomputer checks each line that is entered for syntax. If the ENTER key is pressed and a valid BASIC COMMAND or PROGRAM statement is not on the last line of the screen, the microcomputer will not accept the COMMAND or statement. An extra inverted S cursor will appear somewhere in the last line on the television screen. The inverted S stands for incorrect syntax. If the user cannot determine how to correct the line that is at the bottom of the screen, the entire line can be deleted using the techniques described above. If the line is not corrected or deleted, the syntactically incorrect line will reappear at the bottom of the screen every time the ENTER key is pressed.

Library Functions

A library function is a function that is part of BASIC. Sinclair BASIC has library functions for calculating square roots, logarithms, sines, etc. Functions are printed under each key that has a function. A function is entered by pressing the FUNCTION key while the SHIFT key is held down (Figure 1-4), followed by the key that contains the required function (Figure 1-5). Note that the cursor changes to an inverted F after the FUNCTION key is pressed. The cursor changes back to its previous state after the key with the desired function is pressed. Figure 1-5 shows the functions that can be accessed after holding down the SHIFT key and pressing the FUNCTION key.

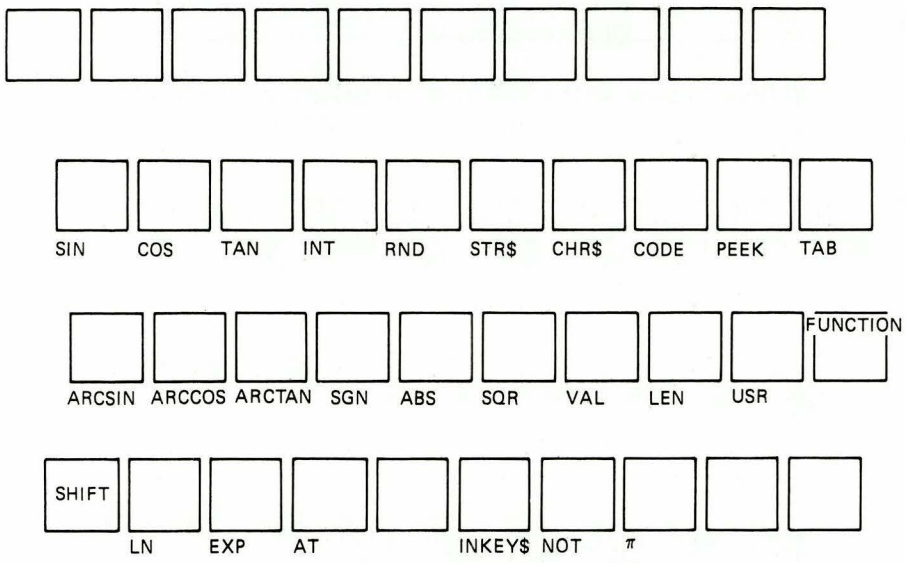


Figure 1-5. Hold the SHIFT key down, then press the FUNCTION key, and then enter one of the above functions.

If the COMMAND

PRINT SQR 2

is entered, the number 1.4142136 will appear at the top left of the screen because 1.4142136 is the square root of 2.

Inverted Characters

If the SHIFT key is held down while the GRAPHICS key (on the 9 key) is pressed, and then one of the letter or number keys in Figure 1-3 is pressed, one will obtain the inverse of the last key that was pressed. The SHIFT and GRAPHICs keys must be released before one of the letter or number keys is pressed. For example, the sequence, SHIFT and GRAPHICS, A, will produce a white A inside of a black space. The computer will stay in the GRAPHICS mode after the first inverted character is entered. Subsequent key depressions will continue to produce inverted characters. One can get out of the inverted character mode by holding the SHIFT key down while pressing GRAPHICS, or just pressing the ENTER key.

Graphics

If the sequence SHIFT, GRAPHICS, SHIFT is entered, a depression of one of the keys will enter the special GRAPHICS symbols or one of the characters shown in Figure 1-6.

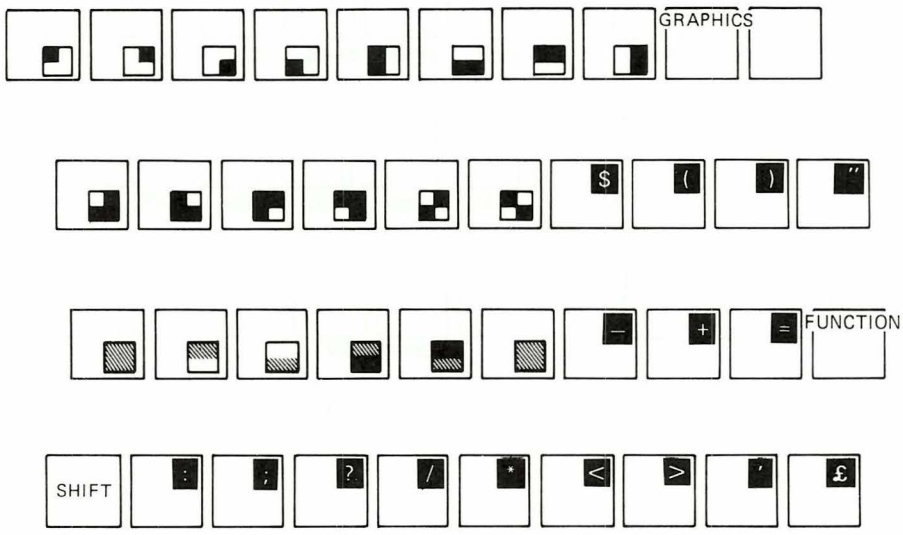


Figure 1-6. Hold the SHIFT key down while pressing GRAPHICS, then press SHIFT and one of the keys to obtain one of the above special GRAPHICS symbols or characters.

After one of the characters in Figure 1-6 is entered the computer will remain in the GRAPHICS mode. A depression of a key will produce an inverted number or letter. A depression of the SHIFT key followed by a depression of another key will produce one of the characters in Figure 1-6. The user can get out of the GRAPHICS mode by holding the SHIFT key down while pressing GRAPHICS.

Reports

When the microcomputer successfully executes a COMMAND the result of the COMMAND is displayed at the upper left of the television screen. The bottom of the screen shows 0/0. The first 0 indicates that no problems were encountered in executing the COMMAND. The second 0 indicates that a COMMAND as opposed to a

statement was executed. The number before the / can be any number 0 through 9 or any letter A through F. Different numbers and letters indicate different problems encountered by the microcomputer. The Sinclair manual lists what each of the reports (0 through 9 and A through F) means. The error reports will be discussed in detail in subsequent sections of this book.

Chapter 2

Hierarchy of Arithmetic Operations

Arithmetic Operations

Table 2-1 shows the arithmetic operations that we are going to use. Note that each arithmetic operator can be entered by pressing the shift key and then pressing the key that has the operator. The ** must be entered by pressing the shift key and then the ** key (shifted H). The ** cannot be entered by pressing the shift key and then pressing the * key (shifted B) twice.

Table 2-1. Arithmetic Operations

BASIC Symbol	Meaning
+	addition
-	subtraction
*	multiplication
/	division
**	exponentiation

Using PRINT in a Command

If the arithmetic operations involve only multiplication and division, the computer performs the operations from left to right. Consider the following example

```
PRINT 3*2/3*4
```

Remember that PRINT is a keyword that is entered with one key stroke. The numbers 3, 2, and 4 are each entered with one key stroke, and * and / are arithmetic operators entered with a shifted B and a shifted V. A COMMAND such as

```
PRINT 3*2/3*4
```

must always be terminated with depression of the ENTER key. Note that one must use the * to indicate multiplication and the / to indicate division. The computer works from left to right, first multiplying 3 by 2 to obtain 6. Then the computer divides 6 by 3 and obtains 2. Finally, it multiplies 2 by 4 to obtain 8.

Algebraic Hierarchy

Consider the following example

```
PRINT 4+3*5-2**3/2*5
```

This example includes addition, subtraction, multiplication, division, and exponentiation. The computer does exponentiations before multiplications, divisions, additions, and subtractions. The computer does multiplications and divisions before it does additions and subtractions. It performs these operations in a left to right order. Thus, the computer "scans" the above COMMAND from left to right and finds the first exponentiation. There is only one exponentiation in the example. This exponentiation is 2**3. The computer raises 2 to the third power and obtains 8. The 8 is substituted for 2**3 and the PRINT statement is now functionally identical to

```
PRINT 4+3*5-8/2*5
```

Next the computer scans from left to right and finds the first multiplication or division. In this case it is a multiplication of 3 by 5. The operation is performed and the product 15 is obtained. In effect, the computer substitutes 15 for 3*5 and internally generates

```
PRINT 4+15-8/2*5
```

The computer again scans from left to right and finds the division of 8 by 2. After performing this division the computer internally generates

```
PRINT 4+15-4*5
```

There is only one more multiplication or division in the problem. The computer multiplies 4 by 5 to obtain 20. At this point the computer has the internal equivalent of

```
PRINT 4+15-20
```

Finally, the computer adds 4 to 15 and subtracts 20 to obtain -1.

In summary, the rules of algebraic hierarchy cause the computer to perform exponentiations first, then multiplications and divisions, and finally additions and subtractions. Multiplications and divisions have the same level on the hierarchy. Additions and subtractions have the same level on the hierarchy, but this level is lower than the level for multiplications and divisions. Operations that are at the same level in the algebraic hierarchy are performed from left to right.

Library Functions

Library functions such as INT, SQR, and ABS are contained in Sinclair BASIC

SQR LIBRARY FUNCTION

The SQR library function calculates a square root. If the computer encounters

```
PRINT SQR 9
```

it will print out 3. The computer calculates the square root of the number 9 when it is given the COMMAND

```
PRINT SQR 9
```

Suppose that the following COMMAND is entered

```
PRINT 4+3*SQR 27
```

Library functions are higher on the algebraic hierarchy than exponentiation, multiplication, division, addition, and subtraction. The computer first calculates the square root of 27 and obtains 5.196152. The PRINT statement then becomes the equivalent of

```
PRINT 4+3*5.1961524
```

Since multiplication is higher on the algebraic hierarchy than addition, the computer multiplies 3 by 5.1961524 to obtain 15.588457. The PRINT COMMAND becomes the equivalent of

```
PRINT 4+15.588457
```

Finally, the computer adds 4 to 15.588457 and prints out 19.588457.

ABS Library Function

The ABS (absolute value) library function calculates the absolute value of a quantity. That is, if the quantity is 0 or positive it remains unchanged. If the quantity is negative, its sign is changed to +. For example

```
PRINT ARS -5.33
```

causes the computer to print out 5.33. An argument is assumed to be positive if it is not preceded by a - sign. Placing a - sign before a number to make it negative is called unary negation. One cannot indicate a number is positive by placing a + sign before the number. A number with no sign preceding the number is assumed to be positive.

INT Library Function

When the computer encounters the INT (integer) library function followed by a positive number, it rounds the number down to the next lower number. Thus, the COMMANDs

```
PRINT INT 9.01
PRINT INT 9.3
PRINT INT 9.5
PRINT INT 9.9
```

would all cause the computer to print out 9. If the computer encounters a negative number as the argument of the INT function, it always rounds to the next lower number. Thus, the COMMANDS

```
PRINT INT -9.1
PRINT INT -9.4
PRINT INT -9.7
PRINT INT -9.9
```

would all cause the computer to PRINT out -10. If the computer encounters an INT library function with an integer as the argument, it prints out the argument. For example

```
PRINT INT 1434
```

will cause the computer to print out 1434.

Summary of Library Functions

Additional library functions are described after parentheses are discussed in subsequent sections of this chapter. All library functions are higher on the algebraic hierarchy than multiplication or division. If more than one library function is in an expression, the library functions will be evaluated from left to right. Thus

```
PRINT 5*SQR 4+INT 4.2
```

is the same as

```
PRINT 5*2+INT 4.2
```

is the same as

```
PRINT 5*2+4
```

is the same as

```
PRINT 10+4
```

is the same as

```
PRINT 14
```

Thus, the computer would print out 14 when it evaluated

```
PRINT 5*SQR 4+INT 4.2
```

Parentheses

Consider the following expressions

a. $5+\left|\frac{3x2}{3}\right|$ b. $(5+3)\times\left|\frac{2}{3}\right|$ c. $\frac{5+(3\times 2)}{3}$

The command

```
PRINT 5+3*2/3
```

would solve the problem in expression (a) above. How could one calculate expressions (b) or (c)? Parentheses are used to alter the order of arithmetic operations. Before the computer checks the algebraic hierarchy for library functions, it searches for parentheses. It goes to the innermost set of parentheses and evaluates what is in the parentheses using the rules for algebraic hierarchy. Consider the COMMAND

```
PRINT (5+3)*2/3
```

This COMMAND has one set of parentheses and the 5+3 is in this set of parentheses. When 5 and 3 are summed, the PRINT statement effectively becomes

```
PRINT 8*2/3
```

Now the computer uses the rules for algebraic hierarchy and prints out 5.3333333. Thus,

```
PRINT (5+3)*2/3
```

will calculate expression (b). Expression (c) can be calculated with

```
PRINT (5+3*2)/3
```

Suppose that the computer encounters

```
PRINT 5*(3+(4+SQR 2))
```

The computer goes to the innermost parentheses and evaluates

```
(4+SQR 2)
```

Algebraic hierarchy within the innermost parentheses causes the computer to take the square root of 2, which is 1.4142136. The number 1.4142136 is then added to 4. The PRINT COMMAND

```
PRINT 5*(3+(4+SQR 2))
```

effectively becomes

```
PRINT 5*(3 + 5.4142136)
```

Next the computer evaluates the expression in the remaining parentheses

```
(3+5.4142136)
```

and obtains 8.4142136. The PRINT COMMAND effectively becomes

```
PRINT 5*8.4142136
```

Finally, the computer performs the multiplication and PRINTs out 42.071068 at the top left of the television screen.

The expression

```
(5 + 3) x (2/3)
```

can be expressed in BASIC as

```
PRINT (5+3)*(2/3)
```

or as

```
PRINT (5+3)*2/3
```

Algebraic hierarchy guarantees that the results for the last two PRINT COMMANDs will be the same (i.e., 5.33333333). The expression

$$\frac{5 + (3 \times 2)}{3}$$

can be expressed in BASIC as

```
PRINT (5+3*2)/3
```

It should be obvious that one must have the same number of left and right parentheses in an expression. The COMMAND

```
PRINT 5*((3*2/3(3+4)+2+4)+7)
```

can be evaluated by the computer because it has 3 left parentheses [(] and 3 right parentheses [)]. The COMMAND

```
PRINT 5*((3*2/3(3+4)+2+4)+7
```

cannot be evaluated because it has 3 left parentheses and only 2 right parentheses. An additional right parenthesis should be placed at the end of the expression. The computer would evaluate the COMMAND

```
PRINT 5*((3*2/(3+4)+2+4)+7)
```

in the following steps

```
1.  5*((3*2/(3  +  4)+  2+ 4)+ 7)
2.   5*((     3*2/7  +  2+ 4)+ 7)
3.  5*((       6/7   +  2+ 4)+ 7)
4.  5*((.85714286    +  2+ 4)+ 7)
5.  5*((2.85714286       + 4)+ 7)
6.  5*(6.85714286              + 7)
7.  5*13.857143
8.  69.285714
```

Note that the computer does not accept implied multiplication. The expression

```
(5 + 2)(4 x 2)
```

must be entered as

```
PRINT (5+2)*(4*2)
```

It cannot be entered as

```
PRINT (5+2)(4*2)
```

Additional Functions

Several library functions can be described at this point. The Sinclair microcomputer can calculate the trigonometric functions SIN, COS, TAN, ASN (arcsin), ACS (arccosine), and ATN (arctan). The argument for SIN, COS, AND TAN must be in radians.

Suppose that the user has an angle in degrees. This angle can be converted from degrees to radians with the formula

$$\text{angle in radians} = \frac{\text{(angle in degrees)} \times \text{PI}}{180}$$

An angle can be converted from radians to degrees with the formula

$$\text{angle in degrees} = \frac{\text{(angle in radians)} \times 180}{\text{PI}}$$

The microcomputer has the number PI built into the BASIC interpreter. The PI can be accessed in the same way that a function is accessed. Note that the keyboard shows π . However, a depression of the π key places PI on the screen. If the user enters

```
PRINT PI
```

the microcomputer will PRINT 3.1415927 at the top of the screen. The COMMAND

```
PRINT TAN (45*PI/180)
```

will PRINT the TAN of 45 degrees.

If the computer user wants to find the arctan of an angle with a TAN of 2.4, he enters

```
PRINT ATN 2.4
```

The computer will print 1.1760052, the arctan in radians. If the result is required in degrees, the user would have to enter

```
PRINT ATN 2.4*180/PI
```

The computer will print 67.380135 which is the arctan in degrees.

The SGN (sign) function will produce a -1 if the argument is negative, 0 if the argument is 0, and 1 if the argument is positive. Thus

```
PRINT SGN -4.5
```

will produce -1

```
PRINT SGN (4.4*SQR 4)
```

will produce +1, and

```
PRINT SGN (4+SQR 16-8)
```

will produce 0.

One can obtain the natural logarithm of a number with the LN function. Thus

```
PRINT LN 4.3
```

will produce 1.458615, the logarithm to the base e (natural logarithm) of 4.3. A logarithm to the base of 10 can be produced by dividing the logarithm of the number to the base of e by the logarithm to the base of e of 10. One could PRINT the common logarithm (base 10) of the number 5 with the COMMAND

```
PRINT LN 5/LN 10
```

The LN function produces the exponent to which e must be raised to obtain a given number. The EXP function takes an exponent and raises e to the power denoted by the exponent. If the COMMAND

```
PRINT EXP 5
```

is entered, the computer will raise e to the fifth power and PRINT 148.41316. Thus, LN is used to take the logarithm to the base e of a number and EXP is used to find the antilogarithm to the base e of a number. One should obtain the original number if she takes the antilogarithm of the logarithm of a number. The COMMAND

```
PRINT EXP LN 5
```

should produce 5 as a result. Similarly

```
PRINT LN EXP 5
```

should produce 5 as a result.

If one has a logarithm to the base of 10, she can obtain the antilogarithm by raising 10 to the power denoted by the logarithm. For example, if 5.271 is the logarithm to the base 10, the COMMAND

```
PRINT 10**5.271
```

will PRINT the antilogarithm to the base 10 of 5.271.

The value of e is 2.7182818. If this number is needed in a calculation, it need not be entered key stroke by key stroke. One can substitute EXP 1 for 2.7182818. For example, the COMMAND

```
PRINT EXP 1
```

will produce 2.7182818.

All numbers are not valid arguments for functions. The computer cannot perform the COMMAND

```
PRINT SQR -2
```

because the square root of a negative number is not a real number. The computer cannot execute the COMMAND

```
PRINT ASN 15
```

because a SIN must be a number between 0 and 1. If

```
PRINT ASN 15
```

is entered in the computer, the computer will respond with an error report of A/0 at the lower left hand corner of the screen. The 0 indicates execution of a COMMAND was attempted and the A indicates an invalid argument was encountered.

Exponential Notation

The numbers that we have used in our examples have been relatively small. Sinclair 8K ROM microcomputers perform calculations to slightly over 9 digits of accuracy. The microcomputer displays a maximum of 8 digits on the television screen. The

microcomputer can display 98421792 or 9.8127658. However, it cannot display the numbers 4798654327 or 567125143.213 because they have more than 8 digits.

The microcomputer must express numbers with more than 8 digits by using exponential notation. For example, the microcomputer could print out the number

6.2374E9

Note the letter E in the number. The E informs the computer user that he must move the decimal point in the number to the right as many times as the number that is to the right of the E (9 in this case). Thus, the number

6.2374E9

is the same as

6237400000

When the microcomputer does not display a decimal point, the decimal point is assumed to be at the right of the least significant digit. That is

6237400

is the same as

6237400.

The number

4.17654E11

is the same as

417654000000

That is, the E11 portion of the number "tells" the user to move the decimal point 11 places to the right. If the number to the right of the E is negative, the decimal point must be moved to the left as indicated by the number to the right of E. Thus

4.2765E-6

is the same as

.0000042765

The computer handles negative numbers with E notation in the same way that it handles positive numbers with the E notation. Thus

-6.26512E9

is the same as

-6265120000

Similarly

-3.2375121E-5

is the same as

-.000032375121

Numbers can be entered in the microcomputer using exponential notation. A user can enter 5×10^7 by entering 5E7. The number 1×10^{25} is entered as 1E25, not E^{25}. Numbers that are greater than 10^{38} cannot be entered because the computer can only handle numbers that are less than 10^{38}. If the computer performs a calculation and produces a number that is greater than 10^{38}, an error report starting with 6 will appear at the bottom left of the screen.

Scientific Notation

If one is familiar with scientific notation, it should be apparent that E stands for exponent. Sinclair microcomputers cannot print out subscripts and superscripts. The E notation tells the user that the number to the left of the E must be multiplied by 10 to the power that is indicated by the number to the right of E. Thus

6.754712E11

is the same as

6.754712×10^{11}

is the same as

675471200000

The number

-4.237121E-4

is the same as

-4.237121×10^{-4}

is the same as

-.0004237121

Consider the following number

427651187653211

The computer must express this number in exponential notation because the number has more than 8 digits. The computer may express the number as

4.2765118E14

If the computer user rewrites the number, he obtains

427651180000000

Exponential notation retains the number of decimal places, and the most significant digits of the number. However, the least significant digits are lost. Suppose that one calculated the following subtraction

```
 427651187653211
-427651187613102
----------------
          40109
```

The result of the calculation is 40109. A microcomputer would convert the two numbers to exponential notation before performing the subtraction. Assume that the microcomputer retains 9 digits in its calculations. One would have

4.27651187E14

for the first number, and

4.27651187E14

for the second number. When the second number is subtracted from the first number, the result is 0. Since 40109 is not equal to 0, valuable information is often lost when the numbers are converted to exponential notation.

Range of Scientific Notation

Scientific notation in the Sinclair 8K ROM microcomputers can express numbers that are as small as 1.0×10^{-38} and as large as 10^{38}. These numbers can be negative or positive. That is the computer can also express -1.0×10^{-38} and -1.0×10^{38}. COMMANDS with exponents greater than 38 will produce a syntax error. Program statements that calculate a number that is greater than 10^{38} will produce an error report starting with 6. Numbers that are less than 10^{-38} or -10^{-38} are interpreted as 0.

No Dividing by Zero

The computer cannot divide by zero. Suppose that the computer is given the COMMAND

PRINT 3*2/0

The computer will respond with an error report of 6/0. This report indicates that the computer found it impossible to divide by 0 in the COMMAND that it was executing.

Suppose one has the following COMMAND

PRINT 4*3/(5*2-10)

Since 5*2-10 is equal to 0, the computer will generate an error message starting with 6.

Unary Negation

Consider the following

PRINT 5*(-3)

The computer will print out the number -15. Note that the sign next to the number 3 does not indicate a subtraction, it is merely the sign of the number. This use of negative sign illustrates unary negation, i.e., unary negation occurs when the - sign represents the sign of a number, but not a subtraction. We have noted that a unary positive number is not accepted by the microcomputer. One cannot enter

```
PRINT 5*(+3)
```

for

```
PRINT 5*3
```

Two operators cannot be placed next to each other.

Chapter 3

Entering Programs

Program Mode

A microcomputer user rarely uses the microcomputer in the COMMAND mode. Computer programs are entered in the PROGRAM mode. In the PROGRAM mode each statement that is entered must start with a number. Enter the following three statements.

```
10  LET X=5
20  PRINT X+3
30  STOP
```

These statements form a small program. Remember to press the ENTER key after each statement is entered.

Note that the computer lists the statements that have been entered after each depression of the ENTER key. When the first statement is entered, it is the only statement shown at the top of the television screen. When the second statement is entered, the first and second statements are listed at the top of the screen. If the user makes a mistake in typing a BASIC statement, the computer may refuse to place the statement in the list of statements. That is, the computer checks each statement for syntax before placing it in the list of statements. If a statement is refused, an inverse S will appear in the bottom line of the screen. This inverse S indicates improper syntax. The bottom line can be deleted or modified in the same way that a COMMAND was modified in Chapter 1.

The number at the beginning of a statement causes the computer to store the statement. The statement is not processed when it is entered. For example, pressing the ENTER key after the statement

```
20 PRINT X+3
```

is entered does not cause a result to be printed out. The computer processes the statements in a program in ascending numerical order. In the example above the statements will be processed in the order: statement 10, statement 20, statement 30. We will discuss how to get the computer to run the program in the section on the RUN COMMAND. Let us examine the program in statements 10, 20, and 30 shown above. The statement that is started with the number 10 is an assignment statement because it contains the keyword LET. The keyword LET informs the computer that an assignment is going to occur. The assignment statement

```
10 LET X=5
```

assigns the number 5 to the variable X. The next statement (20) is responsible for printing out the sum of X + 3 when the program is RUN. Since X is 5, the program will cause the number 8 to be printed out.

RUN Command

A program can be run by entereing RUN, followed by an ENTER. RUN is a COMMAND that causes the computer to run the program. Since RUN is a COMMAND, it is not preceded by a statement number. Entering RUN followed by ENTER causes the preceding program to print out 8. More will be said about the RUN COMMAND after the GOTO keyword is discussed.

STOP Statement

The brief program shown above is terminated by statement 30, a STOP statement. The STOP statement causes the microcomputer to STOP executing the program. Note that the computer prints 9/30 at the lower left hand corner of the screen. The 9 is a report that indicates a STOP was encountered. The 30 is the statement number in which the stop was encountered. After the program is RUN the program listing can be returned to the screen by pressing ENTER.

Later we shall see that BASIC has branching procedures that allow a user to alter the order in which statements are run. A program may have more than one STOP statement. Since the STOP statement prints out a line number, it supplies the user with information about what part of the program was being executed when the STOP statement was encountered.

CONT Command

If the CONT key is pressed after a STOP statement stops program execution, program execution will start in the statement after the STOP statement. Of course, pressing the CONT key will not restart the program if the STOP statement is the last statement of the program. When the STOP statement is the last statement in the program, pressing the CONT key produces a report of 0/0, indicating the program was successfully completed. A depression of the ENTER key after the 0/0 report will return the program listing. Suppose that the STOP statement is eliminated from the preceding program. The program becomes

```
10  LET X=5
20  PRINT X+3
```

When the program is RUN, the value of 5 is assigned to X in statement 10 of the program. Then the sum ov X + 3 (which is 8) appears at the top of the screen. The bottom of the screen displays 0/20, indicating the computer successfully completed all statements in the program after executing statement 20. It is a good practice to terminate a program with a STOP statement. However, the preceding example shows that programs will run properly without a STOP statement. The computer will execute statements until it runs out of statements and reports 0/(last statement number), indicating all statements including the last statement were run properly. The program listing can be regained by pressing ENTER.

Form of Statements

Each statement in BASIC starts with a number and is followed by a keyword. A few statements such as

```
30  STOP
```

are terminated by the keyword. Other statements in BASIC have the body of the statement after the keyword. For example, the body of the statement

```
22  LET X=5
```

is X=5. The keyword in the above statement is LET.

Variable Names

Variable names *must* start with a letter of the alphabet. After the initial letter in the name, any combination of letters and numbers can be used in a variable name. Thus

```
X
A1
HOUSELSBIG
and A1B1C1D1
```

are appropriate variable names. The following would not be appropriate variable names.

```
2DOG
IIT■
CIA?
DI■
```

because they do not start with a letter or because they contain characters that are not ordinary (not inverted) letters and numbers. Sinclair BASIC places no limit on the number of letters and numbers in a variable. Spaces can be placed in a variable name. However, they are ignored by the BASIC interpreter. Thus, once can enter

```
MY NAME IS ALLEN
```

for a variable name. The computer will treat this variable in the same way as

```
MY NAME ISALLEN
```

or

MY NAMEISALLEN

or

MYNAMEISALLEN

Variables in LET Statements

When one uses a LET statement, a variable must appear to the left of the equal sign. The variable is the only thing that can be placed to the left of the equal sign in the body of the statement. For example

```
22 LET MY HOUSE=5.3
23 LET X=(7+1)*Y
24 LET Z=A+B+C
```

are all appropriate LET statements. The following statements are not appropriate LET statements.

```
25 LET IITRI+Y=27
26 LET X+4=23*A
27 LET X+SIGMA+Z=A+B+C
```

Statements 25 and 27 have more than one variable to the left of the euqal sign. Statement 26 has a variable and a constant to the left of the equal sign. Later in this book we shall discuss arrays (numbers entered in ordered tables). Since an element of an array can be represented by a variable, one can have an element of an array to the left of the equal sign in a LET statement. We shall see that array elements take on the form A(I), Q(I), C(I,J), and D(1,5). The following statements would be appropriate LET statements

```
42 LET B(I)=5(X+Y)
43 LET A(I,J)=4*(3+Y)
44 LET C(I,J)=INT (5+2)
```

The meaning of variables such as A(I), Q(I), C(I,J), and D(1,5) will be discussed in the chapters dealing with arrays. We shall also see that string variables (e.g., A$) and a string array [e.g., A$(I,J)] elements can be used on the left side of the equal sign in a LET statement. The meaning of strings and string arrays, like the meaning of numeric arrays, will be left to a later chapter.

Order of Statement Execution

A computer executes statements in numerical order with the lowest number statement executed first. The order in which the statements are entered does not matter. If the statements are entered in the order

```
20 PRINT X+3
30 STOP
10 LET X=5
```

the computer will reorder the statements in ascending numerical order and run the statement numbered 10 first. The listing of statements at the top of the screen is placed in ascending order after each statement is entered.

Choosing Statement Numbers

Our previous examples of programs show that numbers can be skipped when numbering statements. In the last example, statement 10 is followed by statement 20, not statement 11. One could follow statement 10 by statement 11. Consider the following programs

Program 1

```
10 LET X=5
11 PRINT X+3
12 STOP
```

Program 2

```
1 LET X=5
2 PRINT X+3
3 STOP
```

The programs are the same with the exception of the numbers that are used to number statements.

When numbering program statements, the lowest number that can be used is 1; i.e., 0 cannot be a statement number. Programs can start with any starting number less than or equal to 9999. If the program starts with statement 9999, it can only have one statement because the highest number that can be entered as a statement number is 9999. Each statement number must be an integer, and not a fraction.

There is an advantage in leaving unused numbers between statements; e.g., statements 10, 20, and 30 rather than statements 1, 2, and 3. Suppose that one numbers a series of three statements 10, 20, and 30. She can later modify the program by placing statements between each of the existing statements. For example, a statement numbered 25 would run between statements 20 and 30.

Initializing Variables

Sinclair BASIC does not start all possible variables at 0 before a program is RUN. The following program is not possible in Sinclair BASIC.

```
10  LET X=Y+7
20  PRINT X,Y
30  STOP
```

When statement 10 is processed, the computer has no value for variable Y. The computer will produce an error report starting with 2. The program can be changed to

```
5  LET Y=2
10  LET X=Y+7
20  PRINT X,Y
30  STOP
```

When statement 10 is processed, X will be assigned the value of Y plus 7. Since Y was assigned a value of 2 in statement 5, X will be assigned a value of 9 in Statement 10. Statement 5 initializes Y at 2. The variable Y must be explicitly initialized with a statement like statement 10. Of course, Y can be initialized with any number that is not larger or smaller than the maximum or minimum numbers that are allowed by Sinclair BASIC.

Changing Statements

If one wishes to change a statement, all he must do is enter the statement number followed by the new statement, followed by ENTER. A statement can be removed from a program by entering the statement number followed by ENTER.

New Command

Entering a NEW followed by a carriage return "erases" all of the program statements from the computer memory. If the program has been run, the microcomputer assigned values to variables in the program. The NEW COMMAND also "erases" the variables that were assigned. After the NEW COMMAND is executed the microcomputer is in essentially the same state it was in when it was first plugged in. This enables the user to start entering a new program with the knowledge that residual statements from a previous program have been removed.

Clear

Sinclair BASIC has a CLEAR keyword that can be used to remove all variables without removing program statements. In the COMMAND mode CLEAR will clear the microcomputer of any variables that were assigned values while calculations were being performed in the COMMAND mode or while a program was RUN. Enter the following sequence

```
NEW
10  PRINT VARIABLE1
RUN
```

The microcomputer will respond with an error report of 2/10, indicating that VARIABLE 1 had not been assigned a value prior to the initiation of the PRINT statement. Now enter

```
LET VARIABLE1=7
RUN
```

The computer will PRINT 7 at the top left of the screen and respond with an 0/0 report. Now enter

```
CLEAR
RUN
```

Since CLEAR removes all variables, the microcomputer will respond with 2/10 because it cannot PRINT a variable that has not been initialized.

LIST Command

The LIST COMMAND enables the programmer to inspect the statements that have been entered. Entering LIST followed by ENTER will cause the computer to display all of the statements starting with the statement that has the lowest number, then the next lowest number, etc. If the program contains more than 22 statements, the microcomputer will stop listing when 22 statements are on the screen. Entering LIST followed by a number will start the listing at the statement number that follows LIST. If this statement number is not part of the program, the listing will start at the next higher statement number. The microcomputer will place 22 statements on the screen starting with the statement number specified by the LIST COMMAND. For example

```
LIST 123
```

will start the listing at statement 123 (or the next statement number of an actual statement).

Automatic Listings

The automatic listings that occur after each depression of the ENTER key always include the statement that was just entered. If all of the statements in the program can fit in the 22 lines at the top of the screen, an automatic listing will include the entire program listing. The starting statement number for an automatic listing depends on the number of statements in the program and where the last listing started.

Editing a Program

A program statement can be deleted by entering a statement number followed by ENTER. A statement number cannot be entered if the bottom line of the screen contains part of another statement or COMMAND. Any partial statement or COMMAND at the bottom of the screen can be deleted by moving the cursor to the right of the partial statement with → , and then using the DELETE key repeatedly.

Suppose that the user wants to edit a statement that has already been entered. The BASIC interpreter has an internal cursor

(inverted >) that shows the current statement. The automatic program listing usually shows the current statement with this cursor. If the EDIT key (shifted 1) is pressed, this statement will appear at the bottom of the screen. The statement at the bottom of the screen can be modified in the same way that a BASIC COMMAND is modified. When the ENTER key is pressed, the modified statement will replace the original version of the statement. If the number of the statement is modified, pressing ENTER will enter the statement as a new statement in the program. The original statement will remain in the program. The original statement can be removed by typing the statement number followed by ENTER.

If the current statement cursor is not on the screen, one can press the ↓ key (shifted 6) and the cursor will appear next to what is now the current statement. The cursor can be moved to make a different statement the current statement. Pressing ↑ will move the cursor to the next higher BASIC statement. Pressing ↓ will move the cursor to the next lower statement. Repeated depressions of ↓ or ↑ can move the current statement cursor to the statement that must be edited. Repeated depressions of the ↑ key can be used to move the cursor to a statement that is lower than the lowest statement currently on the screen. Repeated depressions of the ↓ key can be used to move the cursor to a statement that is higher than the highest statement currently on the screen.

Suppose that the user wants to EDIT a statement that is a number of statements removed from the current statement cursor location. One can enter LIST followed by the statement number of the statement to be edited. If statement 270 must be edited, enter

```
LIST 270
```

After the ENTER key is pressed the current statement cursor points to statement 270. Press EDIT so that statement 270 can be edited.

The editing capabilities of the microcomputer make it possible to remove quickly a statement or partial statement from the bottom of the screen. Press the EDIT key and the bottom line of the screen will be removed and replaced with the statement that is pointed to by the current statement cursor. It does not matter what this statement is because the ENTER key is pressed immediately after the statement appears at the bottom of the screen. The microcomputer will immediately reinsert this statement in its original location and clear the bottom of the screen.

Statement and Command Length

The microcomputer can only place 32 characters on a line of the screen. Nonetheless, statement and COMMAND length is virtually unlimited. Suppose that a PROGRAM statement or COMMAND is being entered on the bottom line of the screen. After 32 characters are entered, the computer enters additional characters starting at the left of the next line on the screen. Thus

```
10  PRINT (((2*3)+7*2**3)-4*2+A+7+9+2)*7
```

will appear as

```
10  PRINT (((2*3)+7*2**3)-4*2+A+7
+9+2)*7
```

at the bottom of the screen. When ENTER is pressed, the statement will appear as

```
  10  PRINT (((2*3)+7*2**3)-4*2+A
+7+9+2)*7
```

in the program listing. The program listing places fewer characters in the first line of the statement because it automatically reserves four spaces for each statement number.

Chapter 4

Output to the Screen

PRINT Statement for Message

We have already used the PRINT statement as a COMMAND and a PROGRAM statement. Now let us examine the PRINT statement in more detail. Suppose that one wanted to have a program print out a series (string) of words such as ENTER THE NUMBER. The following example shows how this can be accomplished with a PRINT statement.

```
40  PRINT "ENTER THE NUMBER"
```

The statement number is followed by the keyword PRINT. Then the string of words is entered within quotation marks. Any message can be placed within the quotation marks, including messages with numbers and symbols. The only exception to this rule is that quotations marks cannot be used as a symbol within the quotation marks. However, one can use the double quote "" key (shifted Q) to indicate quotation marks inside of the message. When the computer prints the message "" will be printed as ". When the program is RUN, the computer prints out the message exactly as it was entered in the quotation marks. The quotation marks are not included in the output. If the ENTER key is pressed and the string "ENTER THE NUMBER" in the statement

```
40  PRINT "ENTER THE NUMBER"
```

is missing the left or right quotation mark, the computer will not enter the statement. If a right or left quotation mark is missing, the inverse S (syntax) symbol will appear in the statement at the bottom of the screen.

Printing Messages and Numbers Contained in Variables

Note that the statement

```
50 PRINT "X"
```

will print out the letter X when the program is RUN. If the quotation marks are omitted and the statement is entered as

```
50 PRINT X
```

the computer does not print out the letter X. Instead, it prints out the current value for the variable X. The microcomputer will produce an error report of 2/50 if the variable X was not assigned a value earlier in the program. The 2 indicates that X was not assigned a value in the program and the 50 indicates that the user is attempting to use the unassigned variable in statement 50. One can combine messages and variables within a PRINT statement. Examine the following statement

```
45 PRINT "X HAS A VALUE OF ";X
```

If the current value for the variable X is 7.43, the above statement will cause the computer to print

```
X HAS A VALUE OF 7.43
```

Note that a semicolon is placed between the final quotation mark of the message and the variable X. The semicolon causes the computer to print the number 7.43 immediately after the message. The last symbol in the message that we have just discussed is a space (entered by pressing the space key). This space is responsible for the space between the message and the number. If the computer encountered the statement

```
100 PRINT X;Y;Z
```

it would print out the values for the variables X, Y, and Z. However, there would be no spaces betwen the numbers and it would be

impossible to determine where one number started and the next number ended. This problem can be eliminated if the statement is changed to

```
100  PRINT X;" ";Y;" ";Z
```

The X in the PRINT statement causes the value of X to be printed. The space that is enclosed between two quotation marks causes a space to be printed before the value of Y is printed. If the numbers were all positive, they would be separated by one space. A negative number would start with the - sign one space after the last digit of the preceding number.

Assume that the values for X, Y, and Z are 7.43, 7.23456×10^{12}, and 472, respectively. The computer would print out

```
7.43  7.23456E12  472
```

when it encountered the statement

```
100  PRINT X;" ";Y;" ";Z
```

If commas are substituted for the semicolons in the PRINT statement, the computer skips more than one space between numbers. Specifically, the computer assumes that each line on the television is divided into two parts. When the computer encounters a comma, the next number (or message) starts at the beginning of the next unused part. The parts start at the first and sixteenth position on the television screen. It should be noted that Sinclair microcomputers produce 32 columns on the television screen. The commas between numbers guarantee that the largest numbers the computer can generate will be separated by enough space to make a readable output. The statement

```
50  PRINT X,Y,Z
```

would produce the following output

```
7.43      7.23456E12
472
```

The computer prints the third number on the second line of the screen because it can only print two numbers on one line of the screen when the numbers are separated by commas. The statement

```
45  PRINT "X IS",X
```

would produce the following output

```
X IS     7.43
```

The statement

```
48 PRINT "X IS  ";X" Y IS  ";Y
```

will print

```
X IS 7.43 Y IS 7.23456E12
```

Note that the variable X must have a semicolon (or a comma) between the variable and the start of the second message on the line.

Sometimes it is desirable to leave a blank line between two lines on the monitor. Blank lines can make a message or numerical output more readable. A statement number followed by PRINT will produce a blank line. For example, the following program will print out a blank line between two messages

```
10  PRINT "A BLANK LINE FOLLOWS"
20  PRINT
30  PRINT "NOTE THE BLANK LINE"
40  STOP
```

The output for the above program will be

```
A BLANK LINE FOLLOWS

NOTE THE BLANK LINE
```

The same output can be accomplished with

```
10  PRINT "A BLANK LINE FOLLOWS",,,
20  PRINT "NOTE THE BLANK LINE"
30  STOP
```

Since "A BLANK LINE FOLLOWS" takes more than 16 columns on the screen, the first comma in statement 10 moves the print position to the beginning of the next line on the screen. The second comma

moves the print position to the middle of this line, and the third comma moves the print position to the beginning of the next line.

Spaces

The statements that we have used up to this point have had no spaces entered by the user. For example consider the statement

```
10  LET X=5
```

The computer automatically entered a space between the number 10 and the keyword LET. The computer also automatically entered a space between the keyword and the variable X. The user could have inserted a space after the X to produce

```
10  LET X =5
```

The user could have inserted a space after the X and the equal sign to produce

```
10  LET X = 5
```

The computer can recognize symbols such as the equal sign when they are not separated from the rest of the statement by spaces. Spaces make statements easier to read. On the other hand, each space takes up computer memory. If a program is long enough to use available memory, it can be shortened by eliminating some spaces.

One can use more than one space in places where one space is acceptable. For example

```
20  LET X = 5
```

is the same as:

```
20  LET X   =   5
```

Both of the above statements will produce the same output. Of course, the second statement will take up more memory. Sinclair microcomputers even permit spaces in the middle of numbers and spaces in the middle of variables.

```
LET X = 5.2 03    9
```

is the same as

```
LET X=5.2039
```

Any spaces that occur within strings will occur in the output. The statement

```
25 PRINT "THE DOG      IS BLACK"
```

will produce the following output

```
THE DOG      IS BLACK
```

Putting it All Together

Suppose that one wanted to add two numbers and print out the result. The following program would work:

```
10 LET X = 7.2
20 LET Y = 9.1
30 LET Z = X + Y
40 PRINT "ORIGINAL NUMBERS ARE  ";
50 PRINT X;"  AND  ";Y
60 PRINT "SUM = ";Z
70 STOP
```

The first number is assigned to variable X in statement 10. The second number is assigned to variable Y in statement 20. A sum for the two numbers is calculated in statement 30. The sum is placed in variable Z. Statement 40 prints out the string of words

```
ORIGINAL NUMBERS ARE
```

The string of words is immediately followed by the number 7.2 because of the semicolon at the end of statement 40. This semicolon causes the computer to print the variable X on the same line as the words

```
ORIGINAL NUMBERS ARE
```

The first semicolon in statement 50 causes the next string of words, " AND ", to be PRINTed on the same line of the screen. Notice that our first string of words "ORIGINAL NUMBERS ARE " has a

space before the final quotation mark in the string. The second string " AND ", starts and ends with a space. The spaces in the strings provide a space in the output between the string of words and the number that occurs before or after the string of words. Statement 50 prints out the words

```
SUM =
```

followed by the number calculated for the sum. When the program is RUN, the output would be

```
ORIGINAL NUMBERS ARE 7.2 AND 9.1
SUM = 16.3
```

If the semicolon is removed from statement 40, the output would be

```
ORIGINAL NUMBERS ARE
7.2 AND 9.1
SUM = 16.3
```

Input Statement

One problem with the program described above is that the numbers 7.2 and 9.1 occur in LET statements that are part of the program. If the programmer wanted to change a number, he would have to reenter the entire LET statement (line 10, line 20, or both lines). One can use an INPUT statement to stop the microcomputer so that a number can be entered. Examine the program below

```
10  LET Y=5
20  INPUT X
30  LET Z=X+Y
40  PRINT "SUM OF X AND Y IS  ";Z
50  STOP
```

Statement 20 in the above program is an INPUT statement that stops the BASIC interpreter from processing other statements. The input statement makes the microcomputer place an inverted L cursor at the bottom left of the screen. The program does not continue until a number is entered on the keyboard and the ENTER key is pressed. After the ENTER the computer effectively assigns the number that was entered to the variable X. That is, internally the computer makes the equivalent of an assignment statement. Sup-

pose one enters .567 after the INPUT statement stops statement processing. The effective assignment statement is

```
25  LET X = .567
```

The computer runs the next statement (statement 30) after the ENTER. Statement 30 sums the variables X and Y and assigns the sum to Z. Statement 40 causes the microcomputer to output

```
SUM OF X AND Y IS 5.567
```

Now consider the following program

```
10  PRINT "ENTER VARIABLE X"
20  INPUT X
30  PRINT "ENTER VARIABLE Y"
40  INPUT Y
50  LET Z=X+Y
60  PRINT "NUMBERS WERE ";X;" AND ";Y
70  PRINT "SUM =  ";Z
80  STOP
```

Statement 10 causes the computer to print out the string of words

```
ENTER VARIABLE X
```

Statement 20 is an INPUT statement that stops the microcomputer, placing an inverted L cursor at the bottom left of the screen. The program will not continue until a number is entered on the keyboard and the ENTER key is pressed. After the ENTER, the computer assigns the number that was entered to the variable X. Internally the computer makes the equivalent of an assignment statement. Suppose that the number entered for X is 7.2. The internal assignment is equivalent to the assignment statement

```
25  LET X=7.2
```

The computer runs the next statement of the program (statement 30) after the ENTER. Statement 30 causes the computer to PRINT

```
ENTER VARIABLE Y
```

Then the computer encounters an INPUT statement in statement 40. The computer stops until a number is entered. This time the number is assigned to the variable Y. The remainder of the program (statements 50, 60, 70, and 80) calculates the sum of the two numbers and PRINTs out the numbers and the sum.

When the INPUT statement stops the microcomputer, the user does not have to enter a number such as 5 or 2E18. The user can also enter an expression such as

```
4*22
```

The expression will be evaluated and used as a number for the INPUT statement.

Length of PRINT Statement

Sinclair microcomputers can only display 32 characters on a line of the television screen. A PRINT statement such as

```
60  PRINT "THE SUM OF SQUARES TOTAL IS ";L
```

could produce the following output

```
THE SUM OF SQUARES TOTAL IS 2376
542
```

The number 2376542 is broken in two parts. This problem could be avoided by substituting

```
60  PRINT "THE SUM OF SQUARES"
65  PRINT "TOTAL IS ";L
```

for the original statement 60. A number can take no more than 14 spaces on the screen. Some numbers take considerably fewer spaces. The computer can write PRINT statements to insure that they are not broken up by the line length limitations of the microcomputer.

If a PRINT statement ends with a semicolon, the next PRINT statement will start printing immediately after the first PRINT statement. That is the second PRINT statement will print on the

same line of the screen as the first PRINT statement. Consider the following program segment

```
10  PRINT "SS = ";X;", ";
20  LET Z=45
30  PRINT "DF = ";Y
```

If X is 5 and Y is 2, the program will produce the following output

```
SS = 5, DF = 2
```

The semicolon at the end of the statement 10 causes statement 30 to be printed on the same line as statement 10. Suppose X is 8.2347142E+29 and Y is 1.027654E+8. The above program segment will produce the following output

```
SS = 8.2347142E+29, DF = 1.027654
4E+18
```

Note that part of the last number (Y) is printed on a second line because the screen can only hold 32 columns of characters. Now consider what would happen if our program segment had a comma at the end of statement 10. First assume that X is 5 and Y is 2. The program segment is

```
10  PRINT "SS = ";X;" ",
20  LET Z=45
30  PRINT "DF = ";Y
```

The computer will PRINT

```
SS = 5,                DF = 2
```

A comma at the end of a PRINT statement causes the next PRINT statement to start at the next part (print zone) of the screen. If the first PRINT statement terminates output before the sixteenth column of the screen, the next PRINT statement will start at column 16. If the first PRINT statement ends beyond column 16, the second PRINT statement will start on the next line of the screen. If X is 8.2347142E+29 and Y is 1.027654E18, the above program segment will PRINT

```
SS = 8.2347142E+29,
DF = 1.027654E+18
```

Chapter 5

Branching

Unconditional Branch

Consider the following program

```
10  LET X=5
20  GOTO 40
30  LET X=7
40  PRINT "X IS EQUAL TO ";X
50  STOP
```

Although the program is not very useful, it illustrates the GOTO statement. The variable X is assigned the value of 5 in statement 10. Statement 20 causes an unconditional branch to statement 40. That is, after statement 20 is executed, the next statement to be executed is statement 40. Statement 30 would never be executed in the program shown above. The number after GOTO need not be the number of one of the statements in the program. If the number after GOTO is not a statement number, the GOTO statement will cause the program to branch to the next statement number that is higher than the number in the GOTO statement. Thus statement 20 could be

```
20  GOTO 35
```

or

```
20 GOTO 37
```

We shall soon see how a GOTO statement can be used in a practical program.

RUN versus GOTO

When a program is to be RUN, one can type RUN followed by ENTER. This procedure CLEARs the computer of any variables that have been accumulated by RUNning a previous program or by entering COMMANDs. If the user enters RUN followed by a statement number, the variables will be cleared and program execution will start at the statement number after RUN. If the program does not have a statement with this number, program execution will start with the next higher statement number. For example

```
RUN 160
```

will clear the variables and statement execution will start at statement 160 or the next higher statement. If the user wants to start executing a program, but does not want to clear the variables, he can enter GOTO followed bt a statement number. Thus

```
GOTO 160
```

will start program execution at statement 160, but will not clear variables prior to program execution. If GOTO is used to start a program, a statement number must be placed after GOTO.

Two or more BASIC programs can be placed in the computer. For example, the first program can occupy statements 10 through 200, and the second program can occupy statements 300 through 700. The last statement in the first program must be a STOP statement. When the first program is desired, the user enters

```
RUN
```

or `RUN 10`

or `GOTO 10`

When the second program is desired, the user enters

```
RUN 300
```

or GOTO 300

Remember that

RUN

RUN 10

and RUN 300

clear all variables for both programs. GOTO 10 and GOTO 300 do not clear any variables before the program is run. The first program must terminate with a STOP statement to insure that the computer does not start to run the second program after the first program is completed.

Keywords in Commands and Statements

In the previous section we showed how to use GOTO as a COMMAND. We have also used GOTO in a program statement. All keywords except INPUT can be used in PROGRAM statements and COMMANDS. Some keywords such as PRINT and GOTO are often used in PROGRAM statements and COMMANDS. Other keywords such as RUN, and LIST are almost always used in COMMANDS. Other keywords such as IF are almost always used in PROGRAM statements.

BREAK Key

Sometimes a programmer makes a mistake and writes a program that could continue to run without stopping. Consider the following program

```
10 LET X=1
20 GOTO 10
30 STOP
```

The program will assign 1 to X in statement 10. Then statement 20 will send control to statement 10 to assign 1 to X again. Statements 10 and 20 will be repeatedly executed. One can press the BREAK key while the program is running. The BREAK key will stop the program with report D/10 or D/20. The D indicates that the BREAK

key was pressed and the number 10 or 20 is the statement that was being executed while the BREAK key was pressed. The BREAK key can also be used to stop the execution of the SAVE or LOAD cassette COMMANDS that will be discussed in a later chapter.

Relational Operators

Table 5-1 shows relational operators that can be used in BASIC.

Table 5-1. Relational Operators That Can Be Used in BASIC

Relational Operators	Meaning
=	equal to
<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to
< >	not equal to

Note that a given relational operator must be entered with a single key stroke. That is < > is a shifted T and cannot be entered as a shifted N followed by a shifted M.

IF-THEN Statement

Relational operators are most often used in IF-THEN statements. The following program uses an IF-THEN statement in statement 20.

```
10 INPUT X
20 IF X > 5 THEN STOP
30 PRINT X
40 STOP
```

When the program is RUN, INPUT statement 10 places an inverted L cursor on the screen. After the user enters a number, statement 20 causes the computer to determine if the number is greater than 5. If the number is greater than 5, statement 20 makes the BASIC interpreter STOP processing statements. The computer prints 9/20 at the lower left hand corner of the screen. The 9 indicates a STOP was encountered, and the 20 indicates the STOP occurred in statement 20. If the number is less than or equal to 5, the STOP at the end

of statement 20 is ignored and statement 30 causes the value of X to be printed. If the value of X is printed, a 9/40 appears at the lower left side of the screen to indicate that the program terminated in statement 40. The program guarantees that a value of X that is greater than 5 will never be printed. The IF-THEN statement in statement 20 is a conditional branch statement.

Consider the following IF-THEN statement

```
15  IF X=N+1 THEN GOTO 110
```

An IF-THEN statement starts with a statement number followed by the keyword IF. A condition follows the IF. In our present example the condition is X=N+1. After the condition the word THEN occurs. Finally the statement is completed by any acceptable BASIC statement such as GOTO 110. In summary, the IF-THEN statement takes the form of a statement number followed by

IF Condition THEN Statement

e.g., IF X=N+1 THEN GOTO 110

The IF-THEN statement works in the following way. If the condition that follows the IF is met, the statement that follows THEN is executed, otherwise this statement is skipped and the next statement in the program is executed. Suppose that X is equal to 3 and N is equal to 4. The IF-THEN statement and the following statement could be

```
50  IF X=N+1 THEN GOTO 110
60  PRINT "ENTER SCORE ";X
```

Since X is equal to 3 and N is qual to 4, X is not equal to N + 1; i.e., 3 is not equal to 5. The condition in the IF-THEN statement is not met (i.e., X is not equal to N + 1). As a result the statement following THEN is skipped and the program causes statement 60 to be executed. Suppose X is 5 and N is 4. When statement 50 is encountered, X is equal to N + 1. Since the condition in the IF-THEN statement is met, the statement following THEN is executed. In the present example this statement is GOTO 110. This statement causes transfer of control to statement 110 of the program.

The condition that follows the IF in an IF-THEN statement can use any of the relational operators listed in the relational operator section. For example, the condition

```
IF X < N +1
```

tests to determine if X is less than N + 1. IF X is less than N + 1, the statement that follows the THEN is executed, otherwise the program skips to the next numbered statement in the program. The expression to the left of the relational operator need not be an isolated variable. For example

```
IF X+1 < > Y+5
```

is an appropriate condition. If X + 1 is not equal to Y + 5, the statement after THEN is executed. The condition on either side (or both sides) of the relational operator need not contain a variable. For example

```
IF 4 < = Y+2
```

tests if 4 is less than or equal to the sum of the current value of Y + 2. Similarly

```
IF X*X+Y > = 2
```

tests to determine if the current value of the square of X plus the current value of Y is greater than or equal to 2. The rules for algebraic hierarchy apply to the expressions on either side of the relational operator.

Statement after THEN

The statement that follows the THEN in an IF-THEN statement can be any valid BASIC statement. The following are examples of valid IF-THEN statements.

```
200  IF X<5 THEN PRINT "X < 5"
210  IF X< >7+Y THEN LET Q=7+Y
```

Consider the following program segment

```
220  IF X < >5 THEN LET Z=2
230  LET S=5
```

If the condition in the IF-THEN statement is met, the LET Z=2 statement is executed. The next statement to be executed after LET Z=2 is statement 230 (LET S=5). If the condition for the IF-THEN

statement in statement 220 is not met, LET Z=2 is not executed. The next statement to be executed is statement 230 (LET S=5).

AND

Carefully examine the following statement

```
240  IF X>5 THEN IF X<=7 THEN PRINT "OUT OF LIMITS"
```

This statement has a second IF-THEN statement as the statement for the first IF-THEN statement. Statement 240 requires that the conditions in both IF-THEN statements be met for OUT OF LIMITS to be printed. Specifically, any value of X that is greater than or equal to 5 and less than or equal to 7 will cause OUT OF LIMITS to be printed. Sinclair BASIC allows AND to be used in an IF-THEN statement. One could substitute

```
240  IF X>= AND IF X<=7 THEN PRINT "OUT OF LIMITS"
```

for

```
240  IF X>= THEN IF X<=7 THEN PRINT "OUT OF LIMITS"
```

OR

Examine the following program segment

```
10  IF X>7 THEN GOTO 30
20  IF X>=5 THEN GOTO 60
30  PRINT "OUT OF LIMITS"
    ————————————————————
    ————————————————————
60  ————————————————————
```

The program segment illustrates the OR rule. If the value of X is less than 5 OR greater than 7, OUT OF LIMITS is printed. If the value of X is between 5 and 7 (including 5 and 7) control is transferred to statement 60. The dashes in the program segment indicate program statements that are not of interest to us. Sinclair BASIC allows the user to user OR in an IF-THEN statement. The following

statement could be substituted for statements 10, 20, and 30 in the above program segment

```
10  ID X<5 OR X>7 THEN PRINT "OUT OF LIMITS"
```

The program segment becomes

```
10  IF X<5 OR X>7 THEN PRINT "OUT OF LIMITS"
_____________________________
_____________________________
60  _____________________________
```

NOT

Sinclair BASIC allows NOT in a BASIC statement. That is

```
NOT (X>=5)
```

is the same as

```
X<5
```

The NOT negates the expression that follows. Note that

```
10  IF NOT X>=5 OR NOT X<=7 THEN PRINT "OUT OF
LIMITS"
```

is the same as

```
10  IF X<5 OR X>7 THEN PRINT "OUT OF LIMITS"
```

Scroll

Suppose that the following program is RUN.

```
10  PRINT "ENTER NUMBER"
20  INPUT X
30  IF X>22E22 THEN GOTO 80
40  PRINT "NUMBER," "LOGARITHM"
50  PRINT X,LN X/LN 10
60  PRINT
70  GOTO 5
80  STOP
```

The program will calculate the common logarithm for each number that is entered (statement 20) and PRINT the number and common logarithm. If a number that is greater than 22E22 is entered, the program STOPs. Each logarithm that is calculated takes three lines on the screen, one to PRINT the column headings NUMBER and LOGARITHM (statement 40), one to print the number and its logarithm (statement 50), and one to place a blank line on the screen (statement 60). The program fills the television screen from top to bottom. After five logarithms are calculated, 20 lines of the screen are filled. The computer will start to print out the sixth logarithm. It will get as far as PRINTing the heading in statement 40. However, the top of the screen will then be filled with 22 lines. The computer will stop with report 5/40. The 5 indicates that the screen is full and the 40 indicates that statement 40 was the last statement that could be PRINTed. Pressing the CONT key will start the program again. The screen will be cleared and the computer will continue by printing the first unprinted statement. This statement is statement 50 in the present program. The CONT key clears the screen and continues printing in the first unprinted statement. Suppose that the logarithm is modified as follows

```
 5 SCROLL
10 PRINT "ENTER NUMBER"
20 INPUT X
30 IF X>22E22 THEN GOTO 80
35 SCROLL
40 PRINT "NUMBER","LOGARITHM"
45 SCROLL
50 PRINT X,LN X/LN 10
55 SCROLL
60 PRINT
70 GOTO 5
80 STOP
```

The logarithm program now has a SCROLL statement above each PRINT statement. The computer will start printing the program at the bottom of the screen and will move all statements up one position on the screen every time a new statement has to be printed. Numbers can be entered indefinitely. When the top line of the screen is full, the next PRINT statement will make the top line SCROLL off the top of the screen.

One has to be careful about mixing PRINT statements preceded by SCROLL with PRINT statements that are not preceded by SCROLL. If a series of PRINT statements preceded by SCROLL is

followed by a PRINT statement that is not followed by SCROLL, the computer will STOP with a report of 5.

CLS

The CLS (clear screen) COMMAND is useful if one wants to mix SCROLLed PRINT statement with PRINT statements that are not SCROLLed. For example, the program can start with a series of unSCROLLed PRINT statements that use 22 or fewer lines on the screen. Then CLS can be used in a statement to clear the screen. The next section of the program can use SCROLLed PRINT statements.

STOP

Statement 30 in the two preceding programs allows the user to terminate the program by entering any number that is greater than 22E22. When the program stops for INPUT in statement 30, and entry of 23E22 would cause the program to GOTO statement 80 and print a report of 9/80. Sinclair BASIC allows the user to omit statement 30 from the program. If he wants to terminate the program, he can enter STOP (shifted A) when the program is stopped for INPUT. The program will terminate with report D/20. If the CONT key is later pressed, program execution will start at statement 20. Note that the screen is cleared when CONT is ENTERed. All COMMANDS except COPY clear the screen before they are executed. Suppose that the program is stopped by entering STOP when INPUT is requested. The computer user can perform a series of calculations in the COMMAND mode. The user can still enter CONT to restart the program at statement 20 after the calculations are completed. All of the COMMANDs that were performed ended with a report that had a 0 after the slash. The computer ignores all COMMANDs ending in 0 when program execution is restarted with the CONT key.

Chapter 6

Advanced Branching

Summing in a LET Statement

Consider the following program:

```
10  LET X=1
20  PRINT "X = ";X
30  LET X=X+1
40  IF X<4 THEN GOTO 20
50  STOP
```

Statement 30 contains a LET statement that has the variable X on both sides of the equal sign. Placing the variable, in this case X, on both sides of the LET statement, allows one to sum numbers in this variable. Let us follow the order of program execution. Statement 10 contains a conventional LET statement that assigns 1 to the variable X. Statement 20 PRINTs the current value of X. Specifically, the television will display

```
X = 1
```

Statement 30 has an X on both sides of the equal sign. This statement will cause 1 to be added to the present value of X. Since X is 1 when statement 30 is first encountered, X will become 2 (original value of 1 plus 1) when statement 30 is executed. Statement 40 is an IF-THEN statement that tests X to determine if it is less than 4.

Since X is 2, the GOTO 20 in statement 40 causes the program to execute statement 20. Since X is now 2, statement 20 outputs

```
X = 2
```

Statement 30 then takes the current value of X (which is 2) and adds 1. Thus, X is 3 after statement 30 is executed. Since the value of X is less than 4, statement 40 transfers control to statement 20. Statement 20 causes the following output

```
X = 3
```

Then statement 30 adds 1 to the current value of X. Thus, X is 4 after statement 30 is executed. Since X is not less than 4, statement 40 transfers control to statement 50.

Program for Calculating Mean

Consider the following program for calculating a mean of a series of numbers

```
10  LET M=0
20  LET X=1
30  PRINT "ENTER NUMBER OF SCORES"
40  INPUT N
50  IF X=N+1 THEN GOTO 110
60  PRINT "ENTER SCORE ";X
70  INPUT Y
80  LET M=M+Y/N
90  LET X=X+1
100  GOTO 50
110  PRINT "MEAN = ";M
120  STOP
```

First examine statement 80. It is an assignment statement with a variable, M, on both sides of the equal sign. If statement 80 had been entered as

```
80  LET M = Y/N
```

it would assign the current value of Y/N to M every time it was encountered in the program. By placing M on both sides of the = sign, one insures that the previous value of M is changed to the

previous value of M plus the current value of Y/N. When a variable is on both sides of the = sign, one can perform a summation in that variable. Since previous values of the variable are not lost when the variable is encountered on subsequent occasions, one must be careful to initialize the variable before it is encountered for the first time. If a sum is to start at 0, the variable is initialized at 0 with a statement such as

```
10  LET M = 0
```

We will examine statement 80 in more detail after we explain how the computer can enounter this statement more than one time.

The conventional LET statement in statement 10 of the program for calculating a mean starts (initializes) the sum at 0 by setting M, the variable that will produce a sum, at 0. Statement 20 is used to start the variable X at 1. We shall see why this was done as we progress through the program. Statement 30 prints out the message ENTER NUMBER OF SCORES and statement 40 stops the program so that the number of scores can be entered. Note that statement 30 causes the inverted L cursor to occur at the bottom left of the screen. Program execution is halted until a number is entered. Suppose that the program user enters the number 2. After the user types 2 and ENTER, 2 is assigned to variable N. The 2 that was entered indicates that one is calculating a mean for two numbers.

The IF-THEN statement in statement 50 determines whether X is equal to N + 1. Since X is currently 1 (it was assigned 1 in statement 20) and N + 1 is 3 (we assumed that 2 was entered for N in statement 40), X is not equal to N + 1. As a result, the program skips the GOTO 110 statement at the end of the IF-THEN statement. The program continues at statement 60 and outputs

```
ENTER SCORE 1
```

Note that ENTER SCORE is PRINTed on the television screen because it occurs in quotation marks. The number 2 is printed because the variable X is not in quotation marks. Variable X was assigned 1 in statement 20. Statement 70 stops the program to INPUT the first observation. When the observation is entered, it is assigned to the variable Y. Assume that Y (the first observation) is 6. Statement 80 is

```
80  LET M = M + Y/N
```

The current value of M is 0 because M was assigned 0 in statement 10. Thus:

M = 0 + 6/2

M = 3

Examine statement 90

90 Let X=X+1

The value of X was 1 before this statement was encountered. Since X is on both sides of the equal sign, the previous value of X (remember that X was assigned a value of 1 in statement 20) is added to 1. Thus

X=X+1

=1+1

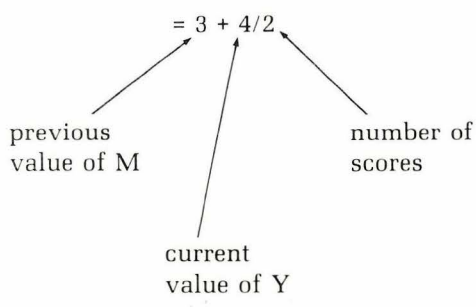
=2

The value for X is 2 after statement 90 is executed. Statement 100 causes the program to branch to statement 50. Statement 50 determines if X is equal to N + 1. Since X is currently 2 (X became 2 in statement 90), and N + 1 is 3, X is not equal to N + 1. The program continues at statement 60 by printing

ENTER SCORE 2

Note that X was incremented in statement 90 so that it would contain the appropriate score number the next time statement 60 was encountered. Assume that the number 4 is entered as INPUT for statement 70. The value of M was 3 when we last encountered statement 80. The statement now evaluates M in the following way

M = M + Y/N



After M in statement 80 is evaluated, its new value is 5.

In statement 90 the number 1 is added to the previous value of X. Thus

```
X=X+1

=2+1

=3
```

Statement 100 causes a branch to statement 50. When statement 50 is processed, X is equal to 3. Since N is 2, N + 1 is 3. For the first time X is equal to N + 1. As a result, the statement after THEN in statement 30 is encountered. This statement is GOTO 110. The program transfers control to statement 110 and PRINTs

```
MEAN = 5
```

Remember that the current value of M is 5. Finally, the program STOPs in statement 120.

Formula for Calculating the Mean

The formula for calculating the mean in statement 80 may seem a bit unusual. Usually we calculate a mean by summing the observations and dividing the sum by N, the number of observations. Of course, one would also obtain the mean if she used the method illustrated by our program for the mean. The examples in Table 6-1 show how both procedures calculate the same mean.

Table 6-1. Two Ways to Calculate the Mean of Numbers 7, 12, and 13

$$M = \frac{7 + 12 + 13}{3} = 10.666 \qquad \text{or} \qquad M = \frac{7}{3} + \frac{12}{3} + \frac{13}{3} = 10.666$$

The program for calculating the mean can easily be modified to sum the numbers and then divide by N after all the numbers are summed. The M in statement 10 must be changed to an S. Statement 80 must be changed to

```
80  LET S = S + Y
```

and statement 110 must be changed to

```
110  PRINT "MEAN = ";S/N
```

The program becomes

```
 10  LET S=0
 20  LET X=1
 30  PRINT "ENTER NUMBER OF SCORES"
 40  INPUT N
 50  IF X=N+1 THEN GOTO 110
 60  PRINT "ENTER SCORE ";X
 70  INPUT Y
 80  LET S=S+Y
 90  LET X=X+1
100  GOTO 50
110  PRINT "MEAN = ";S/N
120  STOP
```

Statement 80 will calculate the sum of the individual observations. Note that the mean is calculated in statement 110 by dividing the sum (S) by the number of observations (N).

Calculations in PRINT Statement

Statement 110 of the last program shows a feature of the PRINT statement that was not discussed in detail when PRINT was first used in the PROGRAM as opposed to COMMAND mode. Calculations can be performed within the PRINT statement for variables that are not within quotation marks. The computer prints out the result of the calculation, not the numbers in the calculation.

Which Program for the Mean?

Now that we have modified the program for calculating the mean, we must decide which version of the program to use. The first version of the program is much better than the revised version of the program. Sinclair microcomputers have a maximum 9 digit capacity for a variable. If the numbers that are averaged are first summed, they can easily exceed the 9 digit by N as it is entered, the larges number that the microcomputer must handle will usually be very close to the mean. It has been noted that one often obtains meaningless results from statistical programs when a computer

reverts to exponential notation in its calculations. The least significant digits that are lost are often very important for accurate results. Any "trick" (such as dividing each observation by N as it is entered) that keeps the numbers in calculations small, helps to insure accurate statistical results.

Power of IF-THEN Statements

The programs that we have devised for calculating the mean are quite versatile. They enable the user to calculate a mean for any number of observations. Let us examine the loop that is used in our program for calculating the mean. Consider the following statements in either program for the mean

```
50  IF X=N+1 THEN GOTO 110
90  LET X=X+1
100  GOTO 50
```

Statement 50 tests variable X to determine if all calculations are complete. The value of X is incremented during each pass through the loop (statement 90). Statement 100 returns control to statement 50 to determine if the incremented value of X exceeds N + 1.

More Than One IF-THEN Statement in a Program

Examine the following program for calculating a series of means

```
1  PRINT "ENTER NUMBER OF MEANS"
4  INPUT W
7  LET Z=1
10  LET M=0
20  LET X=1
30  PRINT "ENTER NUMBER OF SCORES IN GROUP ";Z
40  INPUT N
50  IF X=N+1 THEN GOTO 110
60  PRINT "ENTER SCORE ";X
70  INPUT Y
80  LET M=M+Y/N
90  LET X=X+1
100  GOTO 50
110  PRINT "MEAN ";Z;" = ";M
120  LET Z=Z+1
130  IF Z>W THEN GOTO 150
140  GOTO 10
150  STOP
```

The user enters the number of means that must be calculated. Then the user enters the number of observations for the first mean, followed by the individual observations. After the mean is calculated, the program prints out the mean and requests the number of observations for the second mean. Then the individual observations for the second mean are entered, etc.

The program for calculating a series of means is an extension of the program for calculating a single mean. Examine statements 10 through 110 in the present program and compare them with statements 10 through 110 in our previous (accurate) program for calculating the mean for one set of numbers. The only differences in these statements occur in statements 30 and 110. Statement 30 in the present program prints out the group number (Z) when the number of observations for a group is requested. This was not necessary in our previous program because there was only one group of observations. Similarly, statement 100 in the present program prints out the group number for the mean that was calculated. In summary, statements 10 through 110 in the present program are analogous to the same statements in the accurate program for calculating a single mean. These statements are responsible for calculating one complete mean.

Statement 1 in the present program requests the number of groups for which one is going to calculate means. The number of groups (number of means) is entered in statement 4. This number is placed in variable W. The group number (Z) is initialized at 1 in statement 7.

After the first mean is calculated the number of the group (Z) is incremented in statement 120. The IF-THEN statement in statement 130 tests to determine if the last mean has been calculated; i.e., is the current group number (Z) greater than the actual number of groups (W). If the last group mean has been calculated, the statement after THEN is executed. This statement is GOTO 150. When the program encounters statements 150, it is terminated by a STOP statement. If the question (condition) in statement 130 is not answered in the affirmative, the program skips GOTO 150 and executes the next statement. This statement is

```
140  GOTO 10
```

When the program branches to statement 10, the next mean is calculated.

Chapter 7

Arrays

Introduction to Arrays

BASIC allows a computer user to store numbers in arrays. A single subscripted array is named with a single letter of the alphabet and is followed by an ordinary variable in parentheses. For example, A(I), B(DOG), C(C), and Q(V1) represent single subscripted arrays. A program can have as many as 26 arrays. The first letter (outside of the parentheses) is the array name. The variable inside the parentheses can be any variable that is not being used to store data in the remainder of the program. The variable in parentheses can be changed as the program progresses. Thus, A(I), A(DOG), and A(D1) are the same array.

DIM Statement

Before numbers are entered in an array a DIM (dimension) statement must specify the maximum number of numbers that will be entered in the array. For example, the dimension statement could be

```
10  DIM A(7)
```

The above statement dimensions the array A(I) so that it can contain a maximum of 7 numbers. One does not have to place 7 numbers in the array. Any number of numbers up to 7 can be entered and saved in the array.

Elements of an Array

Suppose that an array is dimensioned as

```
10  DIM A(7)
```

The seven elements in this array can be referred to as A(1), A(2), A(3), A(4), A(5), A(6), and A(7). The numbers in parentheses can be considered to be like subscripts; i.e., A_1, A_2, A_3, A_4, A_5, A_6, and A_7 are analogous to A(1), A(2), A(3), A(4), A(5), A(6), and A(7), respectively.

The DIM statement initializes arrays so that all elements start at 0. That is, if statement 10 DIMensions a 7 element array, statement 20 could be used to PRINT any or all of the array elements. For example, the program

```
10  DIM A(7)
20  PRINT A(2,A(7)
30  STOP
```

would cause two zeros to be printed on the top line of the screen. The elements of the array are set to 0 in statement 10. Elements 2 and 7 are PRINTed by statement 20.

Consider the following program

```
10  DIM A(5)
20  LET I=1
30  PRINT "ENTER A(";I;")"
40  INPUT A(I)
50  IF I=5 THEN GOTO 80
60  LET I=I+1
70  GOTO 30
80  STOP
```

Statement 10 dimensions array A(I) so that it can hold a maximum of 5 numbers. The variable I is assigned 1 in statement 20. Statement 30 is used to PRINT out a request for the number of the array element that is to be entered. The first request from statement 30 is

```
ENTER A(1)
```

Statement 40 contains an INPUT statement that stops program execution until a number is entered. Since I was assigned 1, state-

ment 40 inputs A(1), the first element in the array. Statement 50 tests to determine if I is equal to 5. Since I is equal to 1, the program advances to statement 60 where I is incremented by 1. At this point I is equal to 2 and statement 70 causes the computer to branch to statement 30. Statement 40 outputs ENTER A(2). As a result, the number that is entered is assigned to array element A(2) in statement 40. Statement 50 checks to determine if I is equal to 5. Since I is equal to 2, the program advances to statement 60 and increments I to 3.

The program continues to input numbers into the consecutive elements of array A(I) until the fifth element is entered. Note that the program is not very useful because nothing is done with the elements (numbers) in the array after they are entered. The name of the array A(I) is array A. This array could be referred to as A(J), A(K), etc. The variable in parentheses is a variable name that refers to the number of a given element in array A. The program for entering the elements of the array could have been written as

```
10  DIM A(5)
20  LET Z=1
30  PRINT "ENTER A(";Z;")"
40  INPUT A(Z)
50  IF Z=5 GOTO 80
60  LET Z=Z+1
70  GOTO 30
80  STOP
```

Of course, the variable in parentheses [the Z in A(Z)] must be the same variable as the variable used in the program to hold the number of the current array element. That is, if Z is used in statement 40, it must also be used in statements 20, 30, 50, and 60.

Variables in Dimension Statement

Sinclair BASIC allows a variable in the DIM statement. For example

```
20  DIM B(N)
```

The computer will assume the current value of N when a DIMension statement is processed. Suppose that a program contains the following segment

```
10  PRINT "ENTER NUMBER OF ARRAY ELEMENTS ";
20  INPUT N
30  DIM B(N)
```

If 25 is entered when the computer stops in statement 20, the dimension statement is equivalent of

```
30  DIM B(25)
```

No More Than One Array in DIM Statement

Sinclair BASIC requires that each array in a program is dimensioned in a separate dimension statement. Even though an array can be dimensioned to hold many more elements than will be used, this is not a good procedure. One can easily use up the memory that is available for a program by reserving excessive space for arrays. Many programmers advise placing dimension statements near the beginning of programs. This is important because an array cannot be used before it is dimensioned in the program.

Calculating the Variance

We can use our knowledge of arrays to calculate the variance for a series of numbers. Let us use the formula

$$\text{variance} = \frac{\sum^{I} (X_i - X)^2}{N - 1}$$

where x_i is an observation, X is the mean of the observations, and N is the number of observations. A program to calculate the variance is shown below

```
5   LET X=0
10  PRINT "ENTER NUMBER OF SCORES"
20  INPUT N
30  DIM X(N)
40  LET I=1
50  PRINT "ENTER SCORE ";I
60  INPUT X(I)
70  LET X=X+X(I)/N
80  IF I=N THEN GOTO 105
90  LET I=I+1
```

```
100  GOTO 50
105  LET V=0
110  LET I=1
120  LET V=V+(X(I)-X*(X(I)-X)/(N-1)
130  IF I=N THEN GOTO 160
140  LET I=I+1
150  GOTO 120
160  PRINT "MEAN = ";X
170  PRINT "VARIANCE = ";V
180  STOP
```

When the program is RUN, the variable X is initialized at 0. This variable will be used to calculate the mean. The number of observations that are to be entered is requested in statement 10. This number is entered when the program is stopped at statement 20. The number of observations is assigned to variable N. Statement 30 dimensions array X so that it has N elements. Statement 50 requests an observation. Each time this statement is encountered the number for the requested observation is incremented by 1 in statement 90. The first time statement 50 is encountered, the number (I) is 1 because it was initialized at 1 in statement 40. The elements of the array are entered every time statement 60 stops the program for INPUT. The first time statement 60 is encountered I is equal to 1. As a result, X(I) = X(1) and the first observation is requested.

Statement 70 calculates the mean for the observations. Since X is on both sides of the equation, statement 70 will add each new value of X(I)/N to the previous value of X. Statements 30 and 70 illustrate that the ordinary variable X and the array element X(I) are different variables. Both can occur in the same program. Statement 70 calculates the mean by dividing each observation that is entered by N (the number of observations) and summing this quotient across the entries. The first time statement 70 is encountered I is 1. As a result, X(I)/N = and X(1)/N is equal to the first observation divided by N, the number of observations.

A test in statement 80 determines if all N observations have been entered. That is, has I been incremented enough times (statement 90) so that it is equal to N. If I is not equal to N, I is incremented in statement 90. Then statement 100 causes a branch to statement 50 so that the next observation can be entered. It should be obvious that statements 10 through 80 in the present program use the same technique for entering array elements that was used in the program that illustrated how observations can be entered in

arrays. Additional statements have been added to the entry procedure so that the mean can be calculated as the observations are entered.

When all observations are entered, I is equal to N. This causes statement 80 to execute the GOTO 105 statement. Statements 105 and 110 set V equal to 0 and I equal to 1. A variable such as I that has been used in a part of a program that is completed can be used again in another part of the program. One would not reuse a variable if it contained a result that was needed in later portion(s) of the program. The variable I contains no such result. It was used as a subscript for an array in the first part of the program. Statement 110 reinitializes I (i.e., sets I = 1) so that it can be used as a subscript.

Statement 120 is used to calculate the variance. Remember that the mean of all of our observations has been calculated. The mean is in variable X. The individual obserations are in array X(I). The formula that we are using for the variance is

$$\text{variance} = \frac{\sum^{I}(X_i - X)^2}{N - 1}$$

Let us rewrite our formula for the variance so that it does not require raising $(x_i - X)$ to the second power with an exponent. The microcomputer can obtain a more accurate squared number when the number is obtained by multiplying the number by itself. We obtain

$$\text{variance} = \frac{\sum^{I}(X_i - \bar{X})\ (X_i - \bar{X})}{N - 1}$$

If we substitute X(I) for x_i and X for $\bar{X}$ (these are the variables used in our program for x_i and $\bar{X}$, respectively), we obtain

$$\text{variance} = \frac{\sum^{I}(X(I) - X)\ (X(I) - X)}{N - 1}$$

When this expression for the variance is rewritten in BASIC, it becomes

```
120  LET V=V+(X(I)-X)*(X(I)-X)/(N-1)
```

The * is required in the expression because BASIC does not perform implied multiplications. Parentheses enclose N-1 so that

```
(X(I)-X)*(X(I)-X)
```

is divided by N - 1 and not just N. Of course

```
(X(I)-X)*(X(I)-X)
```

is the square of

```
(X(I)-X)
```

The V on both sides of the = sign (statement 120 of the program for variance) is responsible for summing the squared values of (X(I)-X). The first time statement 120 is encountered, X(I) is equal to X(1), the first observation.

Statement 130 tests to determine if I is equal to N, the number of observations. If I is not equal to N, all of the observations have not been entered in the formula for the variance (statement 120). As a result, I is incremented by 1 (statement 140) and statement 150 causes an unconditional branch to statement 120. Then the next observation in the array is entered in the formula for the variance (statement 120). After the last observation is processed by the formula for the variance, I is equal to N. Statement 130 causes a branch to statement 160 where the mean which is in variable X is printed out. Then statement 170 prints out the variance. Finally, the program is terminated in statement 180.

Computational Formula for Variance

The reader may wonder why we used the difference formula for variance rather than the computational formula. The computational formula would not require storing observations in an array. The computational formula would calculate the variance as follows

$$\text{variance} = \frac{\sum_{1}^{I} X_i^2 - N\bar{X}^2}{N - 1}$$

where x_i is an individual observation, X is the mean of all the observations, and N is the number of observations. The following program uses the computational formula for calculating the variance

```
5   LET V=0
10   LET X=0
20   PRINT "ENTER NUMBER OF SCORES"
30   INPUT N
40   LET I=1
50   PRINT "ENTER SCORE ";I
60   INPUT X1
70   LET X=X+X1/N
80   LET V=V+X1*X1
90   IF I=N THEN GOTO 120
100   LET I=I+1
110   GOTO 50
120   PRINT "MEAN = ";X
130   PRINT "VARIANCE = ";(V-N*X*X)/(N-1)
140   STOP
```

The program has five fewer statements than our previous program and requires no arrays. Individual observations do not have to be stored in memory with the computational formula program for the variance. Nonetheless, the computational formula program is inferior to the difference formula program for the variance. The expression

$$\sum_{I} X_i^2 - N\bar{X}^2$$

involves subtracting one large number (NX^2)) from another large number (Σx_i^2). If the observations involved in the variance calculation are relatively large numbers, the microcomputer may not be able to handle the two large numbers without reverting to exponential notation. The least significant digits of the two numbers will then be lost. These digits are very important in calculating an accurate result when one large number is subtracted from a slightly larger number. Our difference formula program (first variance program) never calculated an intermediate result (or results) that was larger than the final value for the variance. As a result, our first program for the variance is superior to the present program.

How the Computational Formula Program Works

Even though computational formulas should not be used to calculate the variance, let us discuss how the program works. Statements 5 and 10 initialize V, the variance, and X, the mean at 0. The number of observations for the variance are requested in statement 20. This number is entered in statement 50. The variable I is initialized at 1 in statement 40. Statement 50 requests the entry of an observation. For example, the first observation is requested with

```
ENTER SCORE 1
```

The observation is entered in statement 60 and assigned to variable X1. Note that X1 is an ordinary variable and not an array element. The mean is calculated in statement 70 by adding an observation divided by N to the variable X. Since X is on both sides of the equal sign, statement 70 produces a total for all the X/N values that are entered. An observation, X1, is squared in statement 80 and summed to the variable V. Thus, statement 70 calculates the mean, while statement 80 calculates the sum of the squared values for the observations.

Statement 90 checks to determine if all of the observations have been entered; i.e., if I is currently equal to N. If all the observations have not been entered, the variable I is incremented in statement 100. Statement 110 causes a branch to statement 50 so that the next observation can be entered. When all the observations are entered, statement 90 executes the statement after THEN in statement 90 (GOTO 120). Statement 120 prints

```
MEAN =
```

followed by the value that has been calculated for the mean (X). Remember that the computational formula for the variance is

$$\text{variance} = \frac{\sum^{I} X_i^2 - N\bar{X}^2}{N - 1}$$

where Σx_i^2 is the sum of the squared observations, x is the mean of the observations, and N is the number of observations. The value of Σx_i^2 is equal to the value of V that has been calculated in statement

80. The value for X is equal to the mean, X, that has been calculated in statement 70. The N in the formula is the number of observations (N) that was entered in statement 30 of the program. BASIC's equivalent for the above computational variance formula is

```
(V-N*X*X)/(N-1)
```

Notice that the mean squared is X*X and N times the mean squared is N*X*X. Statement 130 in the program outputs

```
VARIANCE =
```

followed by the value calculated for the variance.

Chapter 8

Loops

FOR-NEXT Loops

The programs that we have examined up to this point have used IF-THEN statements when loops were required. Now we shall use FOR-NEXT statements to form loops. A short program using FOR and NEXT statements to calculate a mean is given below

```
 5  PRINT "ENTER NUMBER OF SCORES"
 7  INPUT N
10  LET X=0
20  FOR I=1 TO N
30  PRINT "ENTER SCORE ";I
40  INPUT Y
50  LET X=X+Y/N
60  NEXT I
70  PRINT "MEAN = ";X
80  STOP
```

Notice the statement starting with the keyword FOR in statement 20. Also notice the keyword NEXT in statement 60. The statements between these two statements (i.e., statements 30, 40, and 50) will be run [in order] N consecutive times. The first variable in the FOR statement (in this case I) must be the same variable that is used in

the NEXT statement. Sinclair BASIC requires that the variable following FOR is a single letter of the alphabet. One cannot use variables that are more than one letter long as the first variable in a FOR statement. The program starts with I equal to 1 and then executes statements 30, 40, and 50. Statement 60 increments I to 2 and returns control to statement 20. Statement 20 tests to determine if the current value of I is greater than N. If I is greater than N, the loop is exited and control goes to statement 70, the first statement after the loop. If I is less than or equal to N when statement 20 is executed, statements 30, 40, and 50 are executed in order. Statement 60 then increments I so that I can be tested by statement 20. Note that the FOR statement (FOR I=1 TO N) should be interpreted as FOR I=1 through N. For example, if N is 5, the loop is run five times, for I = 1, I = 2, I = 3, I = 4, and I = 5.

The TO in the FOR statement need not be followed by a variable. TO can be followed by a variable, number, or expression. The following are all valid FOR statements

```
100  FOR I=1 TO 5
110  FOR J=1 TO DOG
120  FOR V=1 TO D+Z+2
```

The expression after TO in the FOR-TO statement is evaluated the first time the FOR-NEXT loop is encountered. The values for the variables that follow TO are the values for these variables when the FOR-NEXT loop is first encountered. Let us return to our first program using a FOR-NEXT loop. Suppose that statements 20 through 60 in the preceding program are changed to

```
20  LET I=1
25  IF I>N THEN GOTO 70
30  PRINT "ENTER SCORE ";I
40  INPUT Y
50  LET X=X+Y/N
55  LET I=I+1
60  GOTO 25
```

The modified program calculates a mean using an IF-THEN statement. Note that the program using the FOR-NEXT loop is shorter than the program using IF-THEN statements. Specifically, the statements

```
20  LET I=1
25  IF I>N THEN GOTO 70
```

```
55  LET I=I+1
60  GOTO 25
```

are replaced with

```
20  FOR I=1 TO N
60  NEXT I
```

The computer user does not have to set up a counter such as

```
55  LET I=I+1
```

when using a FOR-NEXT loop. The BASIC interpreter automatically increments I each time the NEXT I statement is encountered.

One can use any single letter variable in the FOR-NEXT statement. For example, one could have used

```
20  FOR Z=1 TO N
60  NEXT Z
```

However, if Z is used in the FOR statement, Z has to be used in the NEXT statement.

Nesting FOR-NEXT Loops

In this section we shall examine a procedure for nesting FOR--NEXT loops. The program that we used to calculate a series of means is reproduced below

```
  1  PRINT  "ENTER NUMBER OF MEANS"
  4  INPUT W
  7  LET Z=1
 10  LET M=0
 20  LET X=1
 30  PRINT "ENTER NUMBER OF"
 35  PRINT "SCORES IN GROUP ";Z
 40  INPUT N
 50  IF X=N+1 THEN GOTO 110
 60  PRINT "ENTER SCORE ";X
 70  INPUT Y
 80  LET M=M+Y/N
 90  LET X=X+1
100  GOTO 50
```

```
110  PRINT "MEAN ";Z" = ";M
120  LET Z=Z+1
130  IF Z>W THEN GOTO 150
140  GOTO 10
150  STOP
```

A revised version of the program using nested FOR-NEXT loops is shown below

```
 10  PRINT "ENTER NUMBER OF MEANS"
 20  INPUT W
 30  LET M=0
 40  FOR J=1 TO W
 50  PRINT "ENTER NUMBER OF"
 55  PRINT "SCORES IN GROUP ";J
 60  INPUT N
 70  FOR I=1 TO N
 80  PRINT "ENTER SCORE ";I
 90  INPUT Y
100  LET M=M+Y/N
110  NEXT I
120  PRINT "MEAN ";J;" = ";M
130  LET M=0
140  NEXT J
150  STOP
```

The arrows at the left of the program mark off the nested FOR--NEXT loops. The innermost loop is composed of statements 70 through 110. Notice that the variable (I) in the FOR statement (statement 70) is the same variable that is found in NEXT statement 110. The outermost loop (statement 40 through 140) uses J as a variable in statements 40 and 140. A different variable must be used for each loop that is nested in another loop.

Table 8-1. Nested Loops

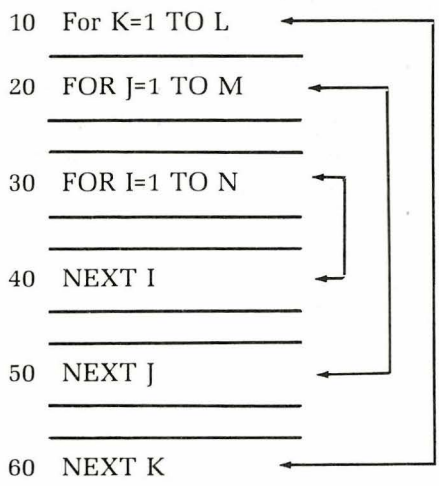
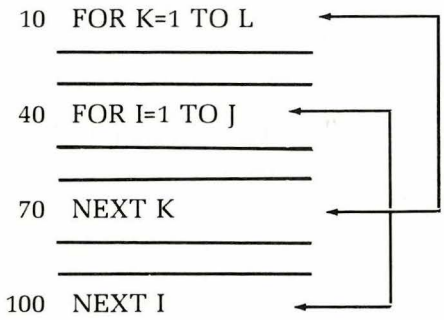


Table 8-1 shows how successive loops can be nested. The dotted lines represent program statements. Notice that the innermost FOR statement uses I as a variable. This means that the innermost NEXT statement must use I for a variable. The next innermost loop uses J in the FOR and NEXT statements. The outermost loop uses K in the FOR and NEXT statements. Inner loops must be contained within outer loops. Table 8-2 shows an example of inappropriate nesting.

Table 8-2. Inappropriate Nesting



Although the I loop starts inside the K loop, it does not end inside of the K loop. That is, the statement NEXT I comes after the statement NEXT K.

Let us examine the program that used nested FOR-NEXT loops to calculate a series of means. Statements 10 and 20 request (statement 10) and INPUT (statement 20) the number of means. This number is placed in variable W so that it can be used as the upper limit for our outer FOR-NEXT loop. The outer FOR-NEXT loop is responsible for the successive means. The other, nested, FOR--NEXT loop is used to calculate a given mean. Statement 30 initializes the variable M at 0. This variable will be used in the calculation of each mean. Statement 40 starts the outermost FOR-NEXT loop (statements 40 through 140). This loop takes on values 1 through W, where W is the number of means to be calculated.

Statements 50 and 55 print out

```
ENTER NUMBER OF
SCORES IN GROUP 1
```

the first time they are encountered. Since these statements are in a loop that goes from 1 through W, they will print out

```
ENTER NUMBER OF
SCORES IN GROUP 2
```

the next time that they are encountered (assuming W, the number of groups, is greater than 1). The number of observations for the requisite group is entered in statement 60 and assigned to the variable N. Statements 70 through 110, the inntermost loop, calculate the mean for a group of observations. That is, a score is entered in statement 90 and assigned to variable Y. The observation is divided by N and then summed to M in statement 100. The innermost FOR-NEXT loop then automatically increments I and returns to statement 70 to determine if all N observations have been entered. If the observations have been entered, the programs goes to statement 120. If N observations have not been entered, the program continues in the loop created by statements 70 through 110.

When the mean is calculated, the inner loop (statements 70 through 110) is exited and statement 120 is processed. This statement PRINTs out the number of the mean and the value of the mean. For example, the third mean that is calculated could produce the following output

```
MEAN 3 = 2.72
```

Statement 130 is used to set M at 0 so that the next mean can be caclulated in variable M. Statement 140 increments the variable J

and causes control to return to statement 40 (the start of the outermost loop). Statement 40 tests to determine if all W means have been calculated. When all means are calculated, the program goest to statement 150 (STOP statement). If all W means have not been calculated, the outermost loop starts the next mean.

Step

A FOR-NEXT statement does not have to be incremented by 1 each time it is encountered. One can explicitly add STEP (shifted E) to a FOR-NEXT statement. The following program will sum 9 numbers.

```
 5 LET X=0
10 LET Y=0
20 FOR I=1 TO 5 STEP .5
30 PRINT "ENTER OBSERVATION"
40 INPUT X
50 LET Y=Y+X
60 NEXT I
70 PRINT "SUM = ";Y
80 STOP
```

The variable I will be incremented by .5 each time the loop is advanced. Values for I will be 1, 1.5, 2, 2.5, 3, 3.5, 4, 4.5, and 5. The

```
FOR = 1 TO 5 STEP .5
```

statement should be thought of as FOR I = 1 through 5 increment I in STEPs of .5.

A STEP can also be negative. If STEP is negative, the number after the control variable must be larger than the number after TO. Examine the following program for adding nine numbers.

```
 5 LET X=0
10 LET Y=0
20 FOR I = 5 TO 1 STEP -.5
30 PRINT "ENTER OBSERVATION"
40 INPUT X
50 LET Y=Y+X
60 NEXT I
70 PRINT "SUM = ";Y
80 STOP
```

I will start at 5 and take on the values 5, 4.5, 4, 3.5, 3, 2.5, 2, 1.5, and 1.

Chapter 9

Multidimensional Arrays

Double Subscripted Arrays

Consider the following numbers

5	7	8	2
4	3	1	9
11	2	1	1

The circled observation is in row 2 and column 4. It can be described as observation 2, 4. Suppose that one assumed all the observations were in a two dimensional array called Q. The circled observation is Q(2,4) where 2 stands for the second row and 4 stands for the fourth column. Observation Q(1,3) is equal to 8. This observation is in the first row and third column. A two dimensional array allows a programmer to specify an observation by its row number and its column number. A two dimensional array, like a one dimensional array, must have a variable name that is one letter of the alphabet. A two dimensional array cannot have a name that was assigned to a one dimensional array.

Dimension Statement

Programs that use double subscripted arrays, like programs that use single subscription arrays, require a DIM statement. The statement

```
10  DIM Q(7,3)
```

would set up an array with a maximum of 8 rows and 4 columns. Each two dimensional array must have its own DIMension statement. An array dimensioned as Z(3,4), could have rows 1, 2, and 3 and columns 1, 2, 3, and 4. Variables can be substituted for one or both of the numbers in the parentheses. These variables can be any valid BASIC variables.

Arrays can have more than two dimensions. An array that has more than two dimensions is difficult to represent on a two dimensional sheet of paper. Consider the following representation of a three dimensional array with dimensions A, B, and C.

		C_1	C_2	C_3	C_4
	B_1	4	7	2	1
A_1	B_2	9	8	2	2
	B_3	7	5	1	1
	B_1	8	7	9	(3)
A_2	B_2	7	1	2	1
	B_3	7	5	4	3

The numbers in the preceding table can be placed in array R(A,B,C). The circled number is R(2,1,4) because it is at the second level of A, the first level of B, and the fourth level of C.

Observations in Double Subscripted Arrays

The following program shows how observations can be entered in a double subscripted array

```
10  DIM P(2,5)
20  FOR J=1 TO 2
30  FOR I=1 TO 5
40  PRINT "ENTER SCORE"
45  PRINT "P(";J;",";I;")"
50  INPUT P (J,I)
```

```
60  NEXT I
70  NEXT J
80  STOP
```

The innermost loop is the I loop. Statement 30 creates 5 columns for the array. Statement 20 creates 2 rows for the array. Let us examine the order in which observations are entered. Statements 20 and 30 start with J and I equal to 1. Thus, P(J,I) starts at P(1,1). Since the innermost loop is the I loop, the next observation that is entered is P(1,2). That is, the innermost loop increments the column numbers. The next observations that are entered are P(1,3), P(1,4), and P(1,5). When P(1,5) is entered, the innermost loop has reached its upper limit of 5. At this point the innermost loop is restarted at 1 (statement 30) and the outermost loop is incremented to 2 (statement 20). Then the observations P(2,1), P(2,2), P(2,3), P(2,4), and P(2,5) are entered. The innermost loop has reached its limit of 5 and the outermost loop has reached its limit of 2. Since both loops are completed, the program goes to the next statement which is a STOP.

Suppose that we want to enter the following numbers in the array program

```
 8     5     6     9     9
12    23    14    19    21
```

These numbers would be entered in order, row by row; i.e., in the order 8, 5, 6, 9, 9, 12, 23, 14, 19, and 21. The computer would make the following array statements:

```
P(1,1)  =  8P(1,2)  =  5  P(1,3)=  6  P(1,4)=  9  P(1,5)=  9
P(2,1)  =  12 P(2,2)=  23 P(2,3)=  14 P(2,4)=  19 P(2,5)=  21
```

Suppose that one wanted to enter the observations by columns rather than by rows (i.e., in the order 8, 12, 5, 23, 6, 14, 9, 19, 9, and 21). The program could be rewritten as follows

```
10  DIM P(2,5)
20  FOR I=1 TO 5
30  FOR J=1 TO 2
40  PRINT "ENTER SCORE"
45  PRINT "P(";J;",";I;")"
50  INPUT P(J,I)
60  NEXT J
70  NEXT I
80  STOP
```

Now the innermost loop is the J loop (the row loop). The first element that is called for is P(1,1). Since the innermost loop is the row loop, the next element requested is P(2,1). At this point the upper limit for the row loop is reached. The row loop is restarted at 1 and the column loop is incremented (statement 20) to 2. Thus, the next two array elements that are requested are P(1,2), and P(2,2). Data from the third, fourth, and fifth columns are entered in the same way as data from the first and second columns.

Printing Row and Column Means

Examine the following program for calculating all row and column means for a double subscripted array.

```
 10  DIM P(2,4)
 20  FOR J=1 TO 2
 30  FOR I=1 TO 2
 40  PRINT "ENTER SCORE"
 45  PRINT "P(";J;",";I;")"
 50  INPUT P(J,I)
 60  NEXT I
 70  NEXT J
 75  LET M=0
 80  FOR J=1 TO 2
 90  FOR I=1 TO 4
100  LET M=M+P(J,I)/4
110  NEXT I
120  PRINT "MEAN ROW ";J;" = ";M
130  LET M=0
140  NEXT J
150  FOR I=1 TO 4
160  FOR J=1 TO 2
170  LET M=M+P(J,I)/2
180  NEXT J
190  PRINT "MEAN COLUMN ";I;" = ";M
200  LET M=0
210  NEXT I
220  STOP
```

The program accepts data from a two subscripted table. Then the program prints out the means for the rows, followed by the means for the columns. Suppose that one uses the following data

```
7    8    2    1
5    6    9    4
```

Statements 10 through 70 enter these observations in a two dimensional array in the same way that was described in the preceding two programs. The observations are entered in the order 7, 8, 2, 1, 5, 6, 9, and 4.

Statements 75 through 140 calculate the means for the rows. Statements 150 through 210 calculate the means for the columns. Note that the row means are calculated by making the columns the innermost loop (statement 90). The observations for a given row are entered into the formula for the mean (statement 100) each time a column subscript is incremented (the row mean does not increment until each number in the row is entered into the formula for the mean).

Column means are calculated by making the row means the innermost loop (statement 160). The observations for a given column are entered in the formula for the mean (statement 170). Each time the row subscript is incremented (the column mean does not increment until each number in the column is entered in the formula for the mean) another observation in entered into the formula for the mean.

Observe that the procedures for calculating row and column means PRINT out the means and set the variable M equal to 0 in steps that are between the inner and outer loops (statements 120 and 130 or statements 190 and 200). This procedure prints out a mean after it is completed (i.e., after the innermost loop is complete) and then sets the variable M equal to 0 for the calculation of the next mean.

More Than Two Dimensions

Sinclair BASIC allows arrays with any number of DIMensions. The array R(DOG,CAT,BAT,MAT) would have four dimensions. An array must always be named with a single letter of the alphabet. No other array can have the same name, even if the other array has a different number of dimensions. Ordinary variables can have the same name as an array name. Each array must be DIMensioned with its own DIM statement. All elements in the array will start at 0.

Chapter 10

Simulating Library Functions

Calculating a Square Root the Hard Way

Sinclair microcomputers have a large number of built in library functions. One of the library functions that is available is the square root (SQR) library function. In this chapter we shall show how one could calculate a square root in a microcomputer system that does not have a SQR function. This procedure for calculating a square root will illustrate how algorithms can be used to calculate functions such as SQR, LN, SIN, etc. More importantly, we shall use the SQR program as a subroutine in the next chapter. A computer can find appropriate algorithms and develop subroutines that can be called by his main program whenever the function defined by the algorithm is required.

A square root can be calculated with the following algorithm

approximation of square root of N = (N/G + G)/2

where N is the original number, and G is initially the "guess" of the square root. After the first time the algorithm is applied, the guess becomes the value of the approximation of the square root of N. We shall use the algorithm to calculate a square root. Suppose that the original number is 25 and the guess of the square root is 7. We have

approximation of the square root of 25 = (N/G + G)/2
= (25/7 + 7)/2
= (3.571 + 7)/2
= 5.285

Our new "guess" is 5.285. Applying the algorithm again we have

$$\begin{aligned} \text{approximation of the square root of } 25 &= (25/5.285 + 5.285)/2 \\ &= (4.730 + 5.285)/2 \\ &= 5.007 \end{aligned}$$

If we applied the algorithm one more time, we would have an almost perfect estimation of the square root of 25.

The algorithm works for positive numbers. Successive "guesses" generated by the algorithm always approach the square root from the same side. For example, if the original number is 25 and the guess of the square root is 7, the next "guess" will be greater than 5 (the actual square root). All successive guesses (until the actual square root is obtained) are greater than 5. If the first "guess" of the square root of 25 was 4, successive guesses would be less than 5. When the guess becomes the actual square root, the algorithm continues to produce the same guess (the square root) every time the square root is entered as a guess. Suppose that one arbitrarily sets the first guess of the square root equal to the original number. For example, the first guess for the square root of 25 would be 25. The program shown below calculates the square root of the number in variable R2. The first guess for the square root is the value of R1.

```
480  PRINT "ENTER NUMBER"
490  INPUT R1
500  LET R3=0
510  LET R2=0
520  IF R1=0 THEN GOTO 610
530  IF R1<0 THEN GOTO 590
540  LET R2=R1
550  LET R2=(R1/R2+R2)*.5
560  IF R2=R3 THEN GOTO 610
570  LET R3=R2
580  GOTO 550
590  PRINT "ERROR SQR"
600  STOP
610  PRINT "SQUARE ROOT OF"
620  PRINT R1;" IS ";R2
630  STOP
```

When the square root is calculated in the above program, it is printed by the statements 610 and 620. Statements 500 and 510 initialize R3 and R2 (the variable that will contain the square root)

at 0. If the original number in R1 is 0, statement 520 causes the program to terminate with R2 at 0. Statement 530 insures that the program never attempts to calculate the square root of a negative number by causing a branch to statement 590 which prints an error message. The first guess of R2 (which is the original number) is generated in statement 540. Statements 550 through 580 form a loop that is not exited until the square root is calculated. Statement 550 calculates an approximation of the square root. If R3 equals R2 (the square root), the same value for R1 was calculated two consecutive times. This means that the approximation of the square root is complete and the loop is exited. If R3 is not equal to R2 in statement 570, the current value of R2 is assigned to R3 and the loop is entered again (statement 580). If

```
555  PRINT R2
```

is added to the preceding program, successive approximations of the square root can be observed.

The program shown below is used when one is sure that a negative number will never be entered for R1. It is three statements shorter than the first square root program.

```
480  PRINT "ENTER NUMBER"
490  INPUT R1
500  LET R3=0
510  LET R2=0
520  IF R1=0 THEN GOTO 580
530  LET R2=R1
540  LET R2=(R1/R2+R2)*.5
550  IF R2=R3 THEN GOTO 580
560  LET R3=R2
570  GOTO 550
580  PRINT "SQUARE ROOT OF"
590  PRINT R1;" IS ";R2
600  STOP
```

Some versions of BASIC have "glitches" in their mathematics packages. Once in a while these versions of BASIC do not produce exactly the same value for R2 and R3 no matter how many times the loop in the square root program is entered. The program shown below is a modification of the previous two square root programs. The new square root program insures that the program gets out of the loop that is used to calculate the square root.

```
480  PRINT "ENTER NUMBER"
490  INPUT R1
500  LET R2=0
510  IF R1<0 THEN GOTO 600
520  IF R1=0 THEN GOTO 570
530  LET R2=R1
540  FOR S=1 TO 25
550  LET R2=(R1/R2+R2)*.5
560  NEXT S
570  PRINT "SQUARE ROOT OF"
580  PRINT R1;" IS ";R2
590  STOP
600  PRINT "ERROR SQR"
610  STOP
```

The above program prevents an infinite loop by using a FOR-NEXT loop for statements 540 through 560. The only disadvantage to this program for calculating a square root is that it takes longer to run than the preceding programs. The FOR-NEXT loop must be executed 25 times to insure an accurate estimate of the square root. If fewer than 25 iterations are used, the microcomputer returns somewhat inaccurate square roots when the initial guess is not close to the square root of the number.

Chapter 11

Using Subroutines

Subroutines

Subroutines are used in situations where a series of steps are repeated in more than one location in a program. For example, consider the following program. The program inputs two numbers and calculates the square root of each number. The program prints out the numbers and their square roots.

```
 10  PRINT "ENTER PAIR OF SCORE"
 20  INPUT R1
 25  INPUT Y
 30  PRINT "FIRST SCORE = ";R1
 40  GOSUB 500
 50  PRINT "SQUARE ROOT OF FIRST"
 55  PRINT "SCORE = ";R2
 60  LET R1=Y
 70  GOSUB 500
 80  PRINT "SECOND SCORE = ";Y
 90  PRINT "SQUARE ROOT OF THE"
 95  PRINT "SECOND SCORE = ";R2
100  GOTO 610
500  LET R3=0
510  LET R2=0
520  IF R1=0 THEN GOTO 590
```

```
530  IF R1<0 THEN GOTO 600
540  LET R2=R1
550  LET R2=(R1/R2+R2)*.5
560  IF R2=R3 THEN GOTO 590
570  LET R3=R2
580  GOTO 550
590  RETURN
600  PRINT "ERROR SQR"
610  STOP
```

GOSUB Statement

In the program shown above statement 40 calls the SQR (square root) subroutine by branching to the number that is placed after the GOSUB keyword. Since the GOSUB statement in statement 40 causes a branch to statement 500, the statement reads

```
40  GOSUB 500
```

The GOSUB keyword, like the GOTO keyword causes an unconditional branch to the program statement that follows the keyword. The GOSUB statement does one thing that the GOTO statement does not do. The GOSUB statement causes the computer to save the location (statement number) of the statement after the GOSUB statement. As a result the computer saves the number of statement 50 which is the next statement after statement 40, the statement that contains GOSUB.

RETURN Statement

The subroutine for calculating a square root is in statements 500 through 600. Notice the RETURN statement in statement 590. Every subroutine must have a RETURN statement so that control can be returned to the main program after the subroutine is completed. Control is always returned to the statement *after* the statement that initiated the subroutine.

Processing the Subroutine

The subroutine is processed until the computer detects a RETURN statement. In this case the RETURN statement is in statement 590. The next statement that is executed is the statement with the number that has been saved (in this case statement 50).

Notice that the program PRINTs out the first number R1, in statement 30. Since R1 is the variable that holds the number for the square root subroutine, R1 has the appropriate value when the GOSUB statement is encountered in statement 40. The square root of R1 is printed out in statements 50 and 55, immediately after the square root is calculated and control is returned to the main program. Statement 60 is used so that the second number, Y, can be assigned to R1, the variable that must hold the number that is used in the square root subroutine. Statement 70 causes a transfer to the square root subroutine so that the second square root can be calculated. Note that statement 70 calls the same subroutine that was used the first time that we encountered a subroutine in the program. The computer saves the number 80, which is the statement number after the current GOSUB statement. When the RETURN in statement 590 is encountered (i.e., after the square root is calculated), control is returned to statement 80, which is the statement after the GOSUB statement that called the subroutine.

Statements 80, 90, and 95 output the second number (statement 80) and the square root (statement 90) of this number. Statement 100 transfers control to the STOP statement so that the subroutine will not be run a third time.

A GOSUB statement, like a GOTO statement, does not require that the number following the keyword be the number of an actual statement. If the number after the keyword is an actual statement number, control branches to this statement. If the number after the keyword is not a statement number, control branches to the next higher statement. Conditional and unconditional branches can occur within a subroutine. The computer will continue to execute the statements as part of the subroutine until a RETURN statement is encountered.

A subroutine approach is useful if a subroutine takes more than two statements (excluding the RETURN). Of course, the subroutine must be called more than one time to make the subroutine approach worthwhile. The more times a subroutine is called, the more useful the subroutine approach.

Subroutines can be nested in subroutines. That is, a subroutine (or subroutines) can be called while another subroutine is being executed. Each subroutine requires a RETURN statement.

Versions of BASIC that have many library functions require fewer subroutines. The Sinclair microcomputers have a SQR library function. The 24 statement program for calculating two square roots can be replaced with

```
 10  PRINT "ENTER PAIR OF SCORES"
 20  INPUT R1
 30  INPUT Y
 40  PRINT
 50  PRINT "FIRST SCORE = ";R1
 60  PRINT "SQUARE ROOT OF FIRST"
 70  PRINT "SCORE = ";SQR R1
 80  PRINT "SECOND SCORE = ";Y
 90  PRINT "SQUARE ROOT OF"
100  PRINT "SECOND SCORE = ";SQR Y
110  STOP
```

Sinclair microcomputers allow the user to address subroutines by name. Consider the following version of the program that calculates square roots using subroutines.

```
  5  LET ROOT=500
 10  PRINT "ENTER PAIR OF SCORES"
 20  INPUT R1
 25  INPUT Y
 30  PRINT "FIRST SCORE = ";R1
 40  GOSUB ROOT
 50  PRINT "SQUARE ROOT OF FIRST"
 55  PRINT "SCORE = ";R2
 60  LET R1=Y
 70  GOSUB ROOT
 80  PRINT "SECOND ROOT =";Y
 90  PRINT "SQUARE ROOT OF"
 95  PRINT "SECOND SCORE = ";R2
100  GOTO 610
500  LET R3 =0
510  LET R2=0
520  IF R1=0 THEN GOTO 590
530  IF R1<0 THEN GOTO 600
540  LET R2=R1
550  LET R2=(R1/R2+R2)*.5
560  IF R2=R3 THEN GOTO 590
570  LET R3=R2
580  GOTO 550
590  RETURN
600  PRINT "ERROR SQR"
610  STOP
```

Statement 5 assigns 500 to variable ROOT. The start of the square root subroutine is a statement 500. Any time the subroutine is required it can be called with

```
GOSUB ROOT
```

or

```
GOSUB 500
```

One can also use the naming technique for GOTO statements. For example

```
100  GOTO FINISH
```

would transfer control to statement 610 if 610 had been assigned to the variable FINISH.

REM Statement

One can place REM statements in a program. These statements will not be processed. For example, the following two statements use REM as a keyword.

```
10  REM THIS IS A PROGRAM
20  REM TO CALCULATE CHI-SQUARE
```

The REMark statements will be ignored when the program is RUN. However, if the LIST COMMAND is initiated, these statements, along with the other statements in the program, will be listed in ascending numerical order. A REM statement helps a computer user remember the purpose of the program or the purpose of the statement or statements following the REM statement. Many programmers place REM statements before each subroutine in a program. The REM statements separate the subroutine from the rest of the program and explain the purpose of the subroutine.

Chapter 12

Slow and Fast Mode

SLOW versus FAST

Sinclair microcomputers are relatively slow. If the COMMAND FAST is entered, the computer will run four times faster than it runs in the SLOW mode. However, the FAST mode has a major disadvantage. No picture is displayed until the program stops for INPUT or until the computer finishes the program. When the computer is plugged in, it is in the SLOW mode. A user can enter SLOW or FAST at any time. The computer will stay in a given mode until the mode is explicitly changed. Always enter long programs in the FAST mode. This will prevent what seems like an interminable wait for a program listing after each statement is entered.

SLOW and FAST can be used as program statements. For example, the first statement in the program could be

```
10  SLOW
```

The statements that follow will be run in the SLOW mode and maintain a steady picture on the screen. After the SLOW portion of the program is run, the next portion of the program may require extensive calculations. A statement such as

```
400  FAST
```

could be used to increase the speed at which subsequent calculations are performed.

When the computer is first turned on, it is in the SLOW mode. If a program is loaded from a cassette, the computer is placed in the same mode (SLOW or FAST) that the computer was in when the program was SAVEd on the cassette.

Chapter 13

Relational Operators in Logical Decisions

Another Use for Relational Operators

BASIC can be used to produce a 1 or a 0, depending on whether an expression is true or false. For example

```
PRINT 15=17
```

will produce a 0 at the upper left of the screen because 15 is not equal to 17. Note that the computer does not PRINT the string

```
15=17
```

because there are no quotes enclosing 15=17. The COMMAND

```
PRINT 22=22
```

will produce a 1 at the upper left of the screen because 22 is equal to 22. The cumbersome expression

```
20  IF (X=15)=1 THEN GOTO 40
```

is the same as

```
20  IF X=15 THEN GOTO 40
```

Examine the first of the two equivalent expressions. If X is equal to 15, then X=15 is 1. If X is not equal to 15, then X=15 is equal

to 0. Since 1=1, but 0 < > 1, control will pass to statement 40 only when X is equal to 15.

Another Use for AND, OR, and NOT

Traditionally AND has meant if condition A is met AND condition B is met, then the result is yes, otherwise the result is no. A 1 can be substituted for yes and a 0 can be substituted for no. The following truth table is a truth table for AND.

Condition		Outcome
A	B	
no	no	0
yes	no	0
no	yes	0
yes	yes	1

The table can be rewritten with 1's for yes and 0's for no.

Condition		Outcome
A	B	
0	0	0
1	0	0
0	1	0
1	1	1

The following program illustrates the AND rule.

```
 10  PRINT "ENTER A? ";
 20  INPUT A
 30  PRINT A
 40  PRINT "ENTER B? ";
 50  INPUT B
 60  PRINT B
 70  PRINT "ENTER NEW A? ";
 80  INPUT AA
 90  PRINT AA
100  PRINT "ENTER NEW B? ";
110  INPUT BB
120  PRINT BB
130  PRINT
140  PRINT "IF ORIGINAL A=NEW A"
```

```
150 PRINT "AND ORIGINAL B=NEW B"
160 PRINT "RESULT IS 1, OTHERWISE 0"
170 PRINT
180 PRINT "RESULT IS ";
190 PRINT A=AA AND B=BB
200 STOP
```

Every time the program is RUN, values are requested for A and B. Then new values are requested for A and B. If the new and old values of A are equal AND the new and old values for B are equal, the program PRINTs 1. In all other cases the program PRINTs 0. The heart of the program is statement 190 which is

```
190 PRINT A=AA AND B=BB
```

A and AA are the old and new values of A. B and BB are the old and new values of B. Statement 190 PRINTs a 1 if A equals AA AND B equals BB. If A < > AA or B < > BB, statement 190 PRINTs 0.

Sinclair BASIC has another kind of truth table for AND.

Condition		Outcome
A	B	
0	0	0
1	0	0
0	1	0
1	1	A

If the AND condition is met, the result is A, not 1. As a result, the statement

```
10 LET LOWER=(A AND A<B)
```

will assign the value of variable A to LOWER when the current value of variable A is less than the current value of variable B. Otherwise LOWER will be assigned 0. The statement

```
10 LET LOWER=(B AND B<A)
```

will assign the value of variable B to LOWER when the current value of variable B is less than the current value of variable A. Otherwise LOWER will be assigned 0. The statement

```
10 LET LOWER=(A AND A<B)+(B AND B<A)
```

will assign the value of B to LOWER when variable B is less than variable A. The expression will assign the value of A to LOWER when the variable A is less than the variable B.

The following program illustrates using AND as described above.

```
 10  PRINT "FIRST NUMBER? ";
 20  INPUT A
 30  PRINT A
 40  PRINT "SECOND NUMBER? ";
 50  INPUT B
 60  PRINT B
 70  PRINT
 80  PRINT "THE TWO NUMBERS ARE";
 90  PRINT A;" AND ";B
100  PRINT "THE SMALLER NUMBER IS "
110  PRINT (A AND A<B)+(B AND B<A)
120  STOP
```

The user enters two numbers and the program PRINTs the smaller of the two numbers.

OR

A conventional OR truth table takes the form

Condition		Outcome
A	B	
no	no	0
yes	no	1
no	yes	1
yes	yes	1

or

Condition		Outcome
A	B	
0	0	0
1	0	1
0	1	1
1	1	1

Examine the following program to illustrate the OR rule.

```
 10  PRINT "ENTER A? ";
 20  INPUT A
 30  PRINT A
 40  PRINT "ENTER B? ";
 50  INPUT B
 60  PRINT B
 70  PRINT "ENTER NEW A? ";
 80  INPUT AA
 90  PRINT AA
100  PRINT "ENTER NEW B? ";
110  INPUT BB
120  PRINT BB
130  PRINT
140  PRINT "IF ORIGINAL A=NEW A"
150  PRINT "OR ORIGINAL B=NEW B"
160  PRINT "RESULT IS 1, OTHERWISE 0"
170  PRINT
180  PRINT "RESULT IS"
190  PRINT A=AA OR B=BB
200  STOP
```

Whenever A equals new A, or B equals new B, or A equals new A AND B equals new B, the program will PRINT 1, otherwise the program will PRINT 0. Note that this program is the same as the program illustrating the AND rule with two exceptions. Statement 150 was changed from

```
150  PRINT "AND ORIGINAL B=NEW B"
```

to

```
150  PRINT "OR ORIGINAL B=NEW B"
```

Statement 190 was changed from

```
190  PRINT A=AA AND B=BB
```

to

```
190  PRINT A=AA OR B=BB
```

Sinclair BASIC has another kind of OR rule that can be summarized as

Condition		Outcome
A	B	
0	0	0
1	0	A
0	1	1
1	1	1

The COMMAND

```
PRINT A OR A < > B
```

will print 1 if A is not equal to B. The same expression will PRINT the value of variable A when A is equal to B. The COMMAND

```
PRINT (A OR A<B)+(B OR >=B)
```

will PRINT the value of A+1 if A>=B, and it will PRINT the value of B+1 when A<B.

The COMMAND

```
PRINT (A OR A<B)+(B OR A>=B)-1
```

will PRINT the value of A when A>=B, and it will PRINT B when A<B.

NOT

The following conventional truth table summarizes NOT.

Condition	Outcome
yes	0
no	1

or

Condition	Outcome
1	0
0	1

Sinclair BASIC uses the same truth table. The COMMAND

```
PRINT NOT (A OR A<B)
```

will produce 0 when (A OR A<B) is evaluated as any number other than 0. The COMMAND will PRINT 1 when (A OR A<B) is evaluated as 0.

Since NOT negates what follows, the statements

```
20  IF A=5 AND B<4 THEN GOTO 120
```

and

```
20  IF NOT (A< >5) AND NOT (B>=4) THEN GOTO 120
```

are equivalent.

Chapter 14

Randomization

RND

The function RND can be used to generate a pseudo-random number that is greater than or equal to zero and less than 1. The pseudo-random number will have eight digits excluding leading 0's. Consider the statement

```
10  LET Y=RND
```

This statement will assign a number that is greater than or equal to 0 and less than 1 to variable Y. The statement

```
10  LET Y=RND*5
```

will assign a number that is greater than or equal to 0 and less than 5 to Y. If statement 10 is changed to

```
10  LET Y=INT (RND*5)
```

the user will obtain one of the integers 0, 1, 2, 3, or 4. One of the integers 1, 2, 3, 4, or 5 could be obtained with

```
10  LET Y=INT (RND*5)+1
```

One die can produce a number between 1 and 6. The result of a toss of one die could be placed in the variable Y with

```
10  LET Y=INT (RND*6)+1
```

If one wanted to represent the sum of the spots on two dice, the statement

```
10  LET Y=INT (RND*6)+1+INT (RND*6)+1
```

could be used.

RAND

The pseudo-random number generator uses a fixed sequence of 65535 numbers. If the program has a statement such as

```
5  RAND 1
```

placed earlier in the program than the first statement using RND, the computer will always start with the same random number. If the statement is changed to

```
5  RAND 2
```

the computer will always start with the same random number, but it will be a different random number than the random number that is supplied for the statement

```
5  RAND 1
```

Any number between 1 and 65535 can be placed after RAND. If the user wants the computer to start with a different random number every time the program is started, the statement

```
5  RAND
```

or

```
5  RAND 0
```

can be placed in the program. In summary, RAND controls the first random number that is generated by RND. This number can be any of 65535 quasi-random numbers.

Examine the following program for generating random integers 1 through N with no replications of any of the integers. When one has such a series of scrambled numbers, it is often referred to as a series of random numbers (integers) without replacement.

```
 10  RAND
 20  PRINT "ENTER NUMBER OF INTEGERS"
 30  INPUT N
 40  DIM A(N)
 50  FOR I=1 TO N
 60  LET Y=INT (RND*N)+1
 70  FOR J=1 TO N
 75  IF A(J)=0 THEN GOTO 100
 80  IF A(J)=Y THEN GOTO 60
 90  NEXT J
100  FOR J=1 TO N
110  IF A(J)< >0 THEN GOTO 150
120  LET A(J)=Y
130  PRINT A(J);" ";
140  GOTO 160
150  NEXT J
160  NEXT I
170  STOP
```

The RAND in statement 10 insures that RND will always start at a different number. Statement 20 requests the number of integers that have to be scrambled. This number is entered in the INPUT statement (statement 30). An array with elements equal to the number of integers is DIMensioned in statement 40. Statements 50 through 160 form the outer FOR-NEXT loop with I as the control variable. There are two FOR-NEXT loops nested in the outer loop. The two nested loops are not nested in each other. For this reason they can both use the same control variable, J. The first of these loops is in statements 70 through 90 and the second is in statements 100 through 150.

Statement 60 generates a random integer that is greater than or equal to 1 and less than or equal to N. The loop in statements 70 through 90 compares the integer that was generated with integers in array A(J). The integers that are generated are placed in array A(J) starting with A(1). After the first integer is entered, all of the remaining integers in the array are 0. Remember that BASIC starts the array with all 0 elements. After the second number is entered only the first two elements will contain elements that are not 0. Statement 75 determines if an array element is 0. If the element is 0, all higher array elements are 0, and it is unnecessary to compare the random integer with any other integers in the array. Control is transferred to statement 100. Statement 80 checks to determine if any array element equals the current random integer. If an array element equals the current random integer, the current integer must

be discarded because it is a replication of an array element. Statement 80 returns control to statement 60 which generates a new random integer.

After a random integer that is different from all the other random integers is generated, the loop examines each array element in succession so that the new number can be placed in the first 0 element of the array. Statement 110 transfers control to statement 150 when the current array element is not 0. Statement 150 increases J so that the next element can be checked to determine if it is 0. When the first 0 element is found in the array, statement 120 places the current random number in this element of the array. Statement 130 PRINTs the random number and PRINTs a space so that the next random number can be printed.

Statement 75 can be omitted from the program. The program will be shorter, but will take longer to RUN. Examine the program to determine why statement 75 makes the program RUN faster.

CLS

The output from the program could be improved by inserting the following statement

```
42  CLS
```

The CLS (clear screen) keyword clears the screen. In the present program it will clear the screen of the words

```
ENTER NUMBER OF INTEGERS
```

before the integers are PRINTed. One could insert

```
44  PRINT "RANDOM NUMBERS (WITHOUT"
46  PRINT "REPLACEMENT) FROM 1 TO ";N
48  PRINT
```

for a nicer heading to the numbers that are PRINTed.

The random number without replacement program places numbers across the lines of the screen. The last number on a line can be confusing because it can be broken up so that part of the number is on one line and part of the number is on the next line. A nicer format can be obtained fro the output if statements 130 and 170 are removed and the following statements are inserted.

```
170  LET J=0
180  FOR I=1 TO N
190  IF J<5 THEN GOTO 220
200  LET J=0
210  PRINT
220  PRINT A(I);" ";
230  LET J=J+1
240  NEXT I
250  STOP
```

THE FOR-NEXT loop in statements 180 through 240 PRINTs the scrambled integers that are in array A(I). The variable J is set to 0 in statement 170. Statement 190 tests J to determine if it is less than 5. Since J starts at 0, statement 190 transfers control to statement 220. The array element is PRINTed in statement 220. Then J is incremented in statement 230 and the array element number is incremented in statement 240. Every time a number is PRINTed J is incremented by 1. When J is equal to 5, statement 190 causes statement 200 to reset J to 0 and execute the PRINT in statement 210. Since the PRINT in statement 210 is not followed by a comma, the computer goes to the next line on the screen before printing the array element. The program PRINTs five numbers on each line of the screen. In the next section we shall learn how to further modify the program so that the numbers form columns on the television screen.

Formatting Output

TAB

The TAB function is used to space the output on a line generated by the PRINT statement. Examine the following statements

```
50  LET X=5.43216
60  LET Y=7.98246
70  PRINT TAB 9;X;TAB 18;Y;
```

Statement 70 will cause the computer to move to the ninth position on the current line of the screen and PRINT the current value of X (starting in the ninth position). Then the computer will move to position 18 on the line and PRINT Y. The output would look like

```
5.43216      7.98246
```

Note that the number after TAB is used to count from the beginning of the line, not from the beginning or end of the last TAB. If the number after TAB would cause PRINTing on top of something that is already on the line, the TAB function starts counting from the beginning of the next line on the screen. For example

```
15  PRINT TAB 7;5.273;TAB 4;4.271
```

would produce

```
5.273
4.271
```

The TAB function will always start at the beginning of the next line rather than back space.

The statement

```
10  LET X=5.2711
20  PRINT TAB 3;"TOTAL";TAB 11;X
```

could PRINT

```
TOTAL     5.2711
```

Each TAB and number is separated from the next variable or string by a semicolon. An expression can be placed in parentheses and used as an argument for a TAB. For example, one could use

```
TAB (X+2)
```

This would cause the computer to start PRINTing X+2 spaces from the beginning of the current line.

In the last chapter we examined a program for generating random integers without replacement. The last version of the program PRINTed 5 numbers per line on the television screen. Statements 170 through 250 in the last version of the program can be replaced with the following statements.

```
170  LET J=0
180  LET K=1
190  FOR I=1 TO N
200  IF J<5 THEN GOTO 240
210  LET J=0
220  LET K=1
```

```
230  PRINT
240  PRINT TAB K;A(I);" ";
250  LET K=K+5
260  LET J=J+1
270  NEXT I
280  STOP
```

The revised program will PRINT the scrambled numbers in 5 columns on the screen.

AT

AT can be used to format PRINT statements. An AT function is followed by two coordinates. The first specifies a line number that is 21 or smaller, and the second specifies a column number that is 31 or smaller. The statement

```
10  PRINT AT 18,1;"NUMBER";AT 7,11;"IS"
```

would PRINT the word NUMBER starting at the first position on line 18 of the screen and IS starting at position 11 on line 7. If the next statement is

```
20  PRINT "TOTAL"
```

it will be PRINTed on line 8, the line after the last line in which output was PRINTed.

PRINT statements that include AT can be directed to PRINT anywhere on the screen. They can be used to PRINT over previous PRINT statements. If a PRINT statement causes the computer to attempt to PRINT beyond line 21 on the screen, an error report starting with 5 will be printed at the lower left of the screen. A depression of the CONT key can be used to continue PRINTing output for the program. The AT function can be used to PRINT anywhere on the screen including lines that have already been used for output.

Chapter 15

Strings

String Handling

Sinclair microcomputers have string handling capabilities that go well beyond the messages that we have placed in quotation marks. A string can be named by a single letter of the alphabet followed by a dollar sign. Thus the statement

```
20  LET Z$="CAT"
```

will assign CAT to Z$. A program can have an ordinary variable named Z, and a string named Z$.

LEN

The length of a string can be determined with the LEN function. Consider the following program.

```
10  PRINT "ENTER A STRING"
20  INPUT Q$
30  PRINT "LENGTH OF ";Q$
40  PRINT "IS ";LEN Q$
50  STOP
```

Statement 10 is used to inform the user that a string is to be entered. Statement 20 stops the program so that a string can be entered. The user does not have to enter the quotation marks because the computer supplies two sets of quotation marks at the

bottom left of the screen. The letters R, O, and M can be entered when the program is RUN. These letters will be placed between the quotation marks. When the program is RUN, statements 30 and 40 will PRINT

```
LENGTH OF ROM
IS 3
```

The words LENGTH OF will be printed because LENGTH OF is in quotation marks. The word ROM will be printed for Q$ because ROM is the string assigned to Q$. IS will be printed because it is in quotation marks. Finally, 3 will be printed because LEN Q$ determines the length of the string.

Run the program again. This time enter

```
THE STRING
```

when a string is requested. The program will PRINT

```
LENGTH OF THE STRING
IS 10
```

The computer will count all of the characters in

```
THE STRING
```

This string contains 10 characters, nine letters, and one space. A string can contain letters, numbers, symbols, inverted letters, inverted numbers, and the special graphic symbols such as ■.

After the program is run with

```
THE STRING
```

entered for the requested string, THE STRING remains in Q$. Run the program again by entering.

```
GOTO 10
```

This time delete the quotation marks when the program stops for a string and enter Q$ followed by ENTER. The quotation marks are deleted by moving the cursor to the right of the line with → and pressing the DELETE key twice. The computer will print

THE LENGTH OF THE STRING
IS 10

Since Q$ was "THE STRING" before the program was run, Q$ can be entered instead of "THE STRING" when a string is requested. If Q$ had not been assigned a string before Q$ was entered, the computer would have stopped with a report of 2/20. If the program is run with RUN or RUN 10 rather than GOTO 10, the computer will CLEAR Q$ and produce a report of 2/20 when Q$ is entered.

VAL

If a string contains a mathematical expression, the computer evaluates the expression with the VAL function. Thus

```
PRINT VAL "7+3/2"
```

will produce 8.5 at the top left of the screen. The COMMAND

```
PRINT VAL "A*7+4+SQR 4"
```

will produce a report starting with 2 unless the variable A was assigned a value prior to entering the COMMAND. If A was assigned 4, the COMMAND

```
PRINT VAL "A*7+4+SQR 4"
```

would PRINT 34.

Later in this book we shall discuss PLOT, and UNPLOT keywords. The function of VAL can only occur in the first of the two coordinates of a PLOT and UNPLOT keyword.

STR$

One can change a number or expression into a string with STR$. Thus

```
10  LET A$=STR$ (7+5)
```

will produce a string that is equal to "12". The statement

```
15  LET C$=STR$(7+A+5)
```

will produce a string that is equal to "18" if A was equal to 6 when the statement was processed. If A was not assigned a value prior to the initiation of statement 15, a report of 2/15 would be produced.

Using VAL and STR$

The VAL function can sometimes save space in a program. Consider the following program.

```
10  LET C$=5*(F-32)/9"
20  LET F$="9*C/5+32"
25  CLS
30  PRINT "THE FORMULA TO CONVERT"
40  PRINT "CENTIGRADE TO FAHRENHEIT"
50  PRINT "IS ";F$
55  PRINT
60  PRINT "THE FORMULA TO CONVERT"
70  PRINT "FAHRENHEIT TO CENTIGRADE"
80  PRINT "IS ";C$
85  PRINT
90  PRINT "IS ORIGINAL TEMPERATURE"
100 PRINT "FAHRENHEIT OR CENTIGRADE ?"
110 PRINT "ENTER F OR C. "
115 INPUT T$
120 PRINT T$
125 IF T$="F" THEN GOTO 180
130 PRINT "ENTER TEMPERATURE CENTIGRADE"
140 INPUT C
145 PRINT
150 PRINT "TEMPERATURE FAHRENHEIT IS"
160 PRINT VAL F$
170 STOP
180 PRINT "ENTER TEMPERATURE FAHRENHEIT"
190 INPUT F
195 PRINT
200 PRINT "TEMPERATURE CENTIGRADE IS"
210 PRINT VAL C$
220 STOP
```

The program can be terminated by entering STOP (shifted A) when the program stops to INPUT the word F or C in statement

110. The two quotation marks at the bottom of the screen must be deleted before STOP can be ENTERed.

Consider the following statement

```
20  Let A$=STR$ (VAL "9+4")
```

The above statement would produce a string that contained "13". The statement

```
25  LET A=VAL (STR$ (9+2+7))
```

would assign 18 to variable A.

CODE

Each letter, number, and symbol used by the microcomputer has a code that is a number between 0 and 255. Enter and RUN the following program

```
10  PRINT "ENTER ONE KEY STROKE"
20  PRINT "FROM THE KEYBOARD"
30  INPUT C$
40  PRINT "THE CODE FOR "C;C$;" IS ";CODE C$
50  PRINT
60  GOTO 10
```

Suppose the letter A (followed by ENTER) is entered when the computer requests

```
ENTER ONE KEY STROKE
FROM THE KEYBOARD
```

The computer will PRINT the number 38 which is the Sinclair code for A. Enter a dash (shifted J) after the next request for input. The computer will respond with 22 which is the CODE for the dash. If the user enters more than one character when the computer requests

```
ENTER ONE KEY STROKE
FROM THE KEYBOARD
```

the CODE function will return the CODE of the first character entered. Keys like EDIT, →, ←, ↑, ↓, GRAPHICS, and DELETE are

used to control the computer. They cannot be used as INPUT for the program. The ordinary quotation marks (shifted P) cannot be ENTERed as INPUT to the program because quotation marks cannot be placed inside of quotation marks. If an inverted character or special graphics symbol is ENTERed, the copmputer must be taken out of the graphics mode with the sequence

SHIFT
GRAPHICS

before ENTER is pressed.

CHR$

A CODE can be entered and CHR$ can be used to print out the character that corresponds to the CODE. Examine the following program

```
10 PRINT "ENTER A CODE (NUMBER"
20 PRINT "FROM 0 to 255)"
30 INPUT A
40 PRINT "CODE IS ";A;" AND CHR$ IS ";CHR$A
50 PRINT
60 GOTO 10
```

The CHR$ function takes a number (CODE) between 0 and 255 and transforms the CODE to a character that can be displayed on the television screen. The computer will not print a CHR$ for some of the keys on the keyboard. For example, the EDIT key and the → key are used to operate the computer, but do not produce characters that can be displayed on the screen. If the computer cannot return a CHR$ for a key depression, it will return a ? The program

```
10 FOR I=1 TO 255
20 PRINT "CODE IS ";I;", CHR$ IS";CHR$ I
30 NEXT I
40 STOP
```

will PRINT each code and its corresponding CHR$. When the screen is filled, the program is continued by depressing CONT followed by ENTER.

INKEY$

Sometimes the computer user wants to have the computer input a single key depression without having to press the enter key after the key is pressed. Suppose that the computer user wants the microcomputer to put a space after each character that is placed on the television screen. The following program can be used to put a space after each character.

```
10  IF INKEY$ < >"" THEN GOTO 10
20  IF INKEY$="" THEN GOTO 20
30  PRINT INKEY$;" ";
40  GOTO 10
```

Note that "" in the statements 10 and 20 are two sets of quotation marks, not the shifted Q.

Statement 10 is used to determine if a key is being held in the down position. If the key is not being pressed, INKEY$ is not equal to "", the empty string. The empty string indicates that no key is being pressed. The GOTO statement in statement 10 keeps executing statement 10 until no key is being pressed. When no key is pressed, statement 20 is executed. The user may not press a key for some time. The GOTO statement in statement 20 causes this statement to be repeatedly executed until a key is pressed. Then INKEY$ < >"" and statement 30 is executed.

Statement 30 prints the CHR$ for the key that is pressed, followed by a space. The semicolon at the end of the statement 30 insures that the next key stroke will be printed on the same line of the screen. Then statement 40 transfers control to statement 10. Note that statement 30 prints the CHR$ and space very rapidly, presumably more rapidly than a finger can be removed from a key. When statement 10 is encountered, this statement is executed until the user's finger is removed from the key. Statement 20 is executed until a key is pressed. After a key is pressed, statement 30 prints this key stroke followed by a space.

A depression of the SPACE/BREAK key will be interpreted as a BREAK in the above program and terminate program execution. How can we enter spaces from the keyboard? We can use a depression of the 0 key to simulate a space. If we want a number 0 on the screen, a letter O can be substituted. The modified program is shown below

```
10  IF INKEY$< >"" THEN GOTO 10
20  IF INKEY$="" THEN GOTO 20
30  IF INKEY$="0" THEN GOTO 60
40  PRINT INKEY$;" ";
50  GOTO 10
60  PRINT "  ";
70  GOTO 10
```

The above program will PRINT a space every time a 0 is entered. That is, statement 30 checks for the 0 and causes statement 60 to print two spaces whenever a 0 is encountered.

Suppose that statement 10 is left out of one of the INKEY$ programs. The program will repeatedly print out the key that is depressed for as long as the key is depressed. If the Z key is depressed, the program will print out Z's with spaces between the Z's for as long as the Z key is depressed.

Suppose that the value of INKEY$ changes while statement 30 is being executed. Then statement 40 outputs a character that is different from the character that was in INKEY$ when statement 30 was first encountered. This problem can be eliminated by changing the program to

```
10  IF INKEY$< >"" THEN GOTO 10
20  IF INKEY$="" THEN GOTO 20
30  LET A$=INKEY$
40  IF A$="0" THEN LET A$=" "
50  PRINT A$;" ";
60  GOTO 10
```

The above program introduces a new problem. The addition of statement 30 makes the program take longer to run. This increases the chance that a new key will be pressed while statements 30, 40, 50, and 60 are being run. Key strokes that occur during the execution of these statements will be undetected.

Concatination

Sinclair microcomputers allow two strings to be concatinated (combine). The statement

```
500  LET C$=A$+B$
```

would create a string, C$, that started with A$ and ended with B$.
If A$ is

"THE SUM OF"

anb B$ is

"THE SQUARES IS"

C$ would be

"THE SUM OF SQUARES IS"

The + sign is used to concatinate strings. However, the -, *,/, and ** symbols cannot be used in string concatination operations.

Chapter 16

Graphics in Strings

Comparing Strings

The IF-THEN statement can be used to compare strings. Consider the statement

```
10  IF "A?B■">"ABC" THEN GOTO 50
```

Sinclair BASIC compares the characters in the string from left to right. That is, the first character in the first string is compared to the first character in the second string, the second character in the first string is compared to the second character in the second string, etc. If two characters are the same, they are considered to be equal. If the first character in the first string has a higher code than the first character in the second string, the first string is higher than the second string. If the first character is the same in both strings, the second characters are compared, etc. The computer user can use any of the relational operators =, <, >, < >, <=, and >= in an IF-THEN statement comparing strings. The strings in the example shown above are equal in the first character (A is equal to A). A "?" has a code of 15 and a "B" has a code of 39. Thus, the first string is less than the second string in the second character. As soon as a character is found that is less than or greater than the corresponding character in the other string, comparisons stop. One knows that

```
A?B■<ABC
```

Since the condition in the IF-THEN statement is not met, control is not transferred to statement 50. Instead control is transferred to the

statement after statement 10. Consider the following examples that show the "lower" of two strings.

JUMP < JUMPED
BOG < HUB■
1 < A

The computer can be used to arrange integers in order or to alphabetize words. Codes for numbers are in ascending order (28 through 37) and codes for letters of the alphabet are in ascending order (38 through 63).

Special Graphics Symbols

There are 22 special graphics symbols. All graphics symbols (with one exception) are obtained by entering

SHIFT
GRAPHICS

and the desired graphics symbol key. All of the symbols except two are shown on the keys.

The ■ is obtained by entering

SHIFT
GRAPHICS

followed by a depression of the SPACE key. The □ symbol is obtained by pressing SPACE and is not preceded by

SHIFT

Graphics

Symbols ▙, ▟, ▛, ▜, ▚, ▞, ▖, ▗, ▘, ▝, ▌, ▐, ▀, ▄, ■, and □ are often used together because they represent all of the ways that a square composed of four small squares can be divided into black and white segments.

Symbols ▄, ▀, ▒, ▂, ▓, ▃, ▒, ■, and □ represent all of the ways a square can be divided in a top and bottom half with a given half white, black or gray. These symbols are useful for bar graphs that go from left to right on the screen. Symbols ▁, ▓, ▒, ▃, ▁, and ▄ go together because they have a dot pattern for at least half of the

symbol. Note that symbol ▩ is different from symbol ⊠. The first of these symbols can be placed next to □ or ▢ to obtain ▩▩ or ▩▢ and the second of these symbols can be placed next to ▣ or ▪ to obtain ⊠▣ or ⊠▪. Consider the following program.

```
10  FOR I=11 TO 21
20  FOR J=6 TO 16
30  PRINT AT J,I;"⊠"
40  NEXT J
50  NEXT I
60  STOP
```

The program will print a large square on the screen. The inner, J, loop starts by printing the first column of the square in column 11 (I was set to 11 in statement 10), starting in row 6 and finishing in row 16. Then I is advanced to 12 and the second column of the square is printed.

Let us consider how little white squares can be printed in the big square. The little squares could be used as eyes for a face. Similarly, little gray squares could be used for ears and two rows of white squares could be used as a mouth.

First we shall make the preceding program into a subroutine with variable A substituted for 11, B substituted for 21, C substituted for 6, and D substituted for 16. The symbol ⊠ will be represented by A$. The following program is the modified program for printing a square.

```
 10  LET A=11
 20  LET B=21
 30  LET C=6
 40  LET D=16
 50  LET A$="⊠"
 60  GOSUB 60
 70  STOP
600  FOR I=A TO B
610  FOR J=C TO D
620  PRINT AT J,I;A$
630  NEXT J
640  NEXT I
650  RETURN
```

The following figure shows what our head will look like when it is completed.

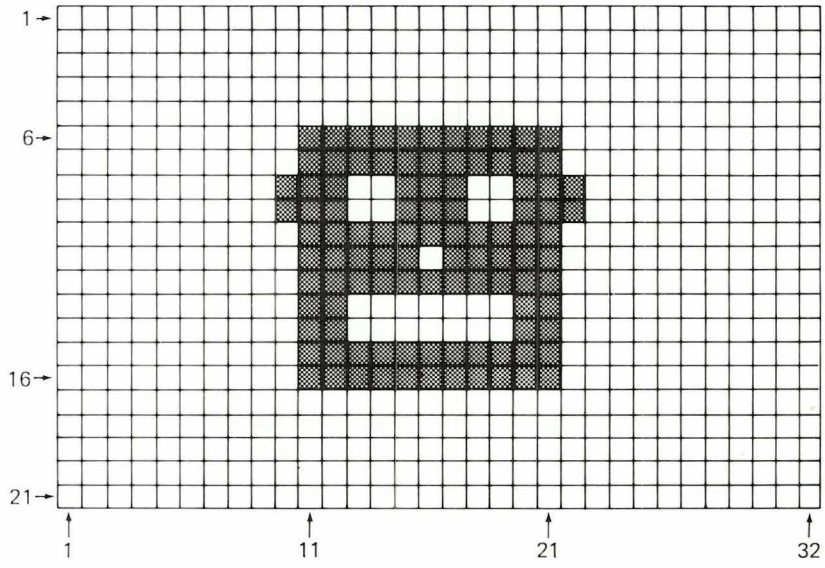


Figure 16-1. The Head.

Now we can remove statement 70 from the square program and have the program print a left eye in the square by adding

```
 70  LET A=13
 80  LET B=14
 90  LET C=8
100  LET D=9
110  LET A$=" "
120  GOSUB 600
130  STOP
```

The right eye can be printed by removing statement 130 and inserting

```
130  LET A=18
140  LET B=19
150  GOSUB 600
160  STOP
```

Note that variables C, D, and A$ already contain the correct numbers for printing the right eye.

A nose can be printed by removing statement 160 and adding

```
160  LET A=16
170  LET B=A
180  LET C=11
190  LET D=C
200  GOSUB 600
210  STOP
```

A left ear can be printed by removing statement 210 and adding

```
210  LET A=10
220  LET B=A
230  LET C=8
240  LET D=9
250  LET A$="□"
260  GOSUB 600
270  STOP
```

A right ear can be printed by removing statement 270 and adding

```
270  LET A=22
280  LET B=A
290  GOSUB 600
300  STOP
```

A mouth can be printed by removing statement 300 and adding

```
300  LET A=13
310  LET B=19
320  LET C=13
330  LET D=14
340  LET A$=" "
350  GOSUB 600
360  STOP
```

One eye can be made to blink by removing statement 360 and inserting

```
360  FOR I=1 TO 200
370  PRINT AT 8,13;"■"
380  PRINT AT 8,13;"□"
390  NEXT I
400  STOP
```

Vertical Bar Graphs

Now we shall examine the program for vertical bar graphs

```
 10 LET A=20
 20 LET B=7
 30 LET K=0
 40 PRINT "ENTER NUMBER OF BARS"
 50 INPUT BARS
 60 IF BARS>6 THEN GOTO 40
 70 DIM A(BARS)
 80 FOR I=1 TO BARS
 90 PRINT "ENTER HEIGHT OF BAR ";I
100 INPUT A(I)
110 IF A(I)>K THEN LET K=A(I)
120 NEXT I
130 CLS
140 PRINT AT 1,1;K;AT 5,1;K*.75;AT 10,1;K/2;
    AT 15,1;K.25;AT 20,1;0
150 FOR I=1 TO BARS
160 FOR J=20 TO (21-INT (A(I)*20/K)) STEP -1
170 IF I=1 THEN LET A$="■"
180 IF I=2 THEN LET A$="■"
190 IF I=3 THEN LET A$="■"
200 IF I=4 THEN LET A$="■"
210 IF I=5 THEN LET A$="■"
220 IF I=6 THEN LET A$="■"
230 PRINT AT A,B;A$
240 LET A=A-1
250 NEXT J
260 LET B=B+4
270 LET A=20
280 NEXT I
290 PRINT AT 21,7;"  ";
300 FOR I=1 TO BARS
310 PRINT I;"   ";
320 NEXT I
330 STOP
```

The program will print 1 to 6 bars on a graph. In order to make more use of the screen, the largest BAR will take 20 lines on the screen. The units for the bars will be printed along the Y axis at the

left of the screen, and the bars will be labeled 1 through 6. Only positive numbers can be entered for the height of a given bar.

Statements 10 and 20 initialize A at 20 and B at 7 for the part of the program that prints out the bars. Statement 30 initializes K at 0. The variable K will eventually contain the height of the highest bar. The number of bars is requested in statement 40 and entered in statement 50. Since the number of bars cannot exceed 6, statement 60 returns control to statement 40 when the user enters a number that is greater than 6. Statement 70 DIMensions array A at 6, the number of bars. The FOR-NEXT loop in statement 90, INPUTs this height in the elements of array A (statement 100), and checks each entry to determine if it is the largest entry up to that point (statement 110). If the entry is larger than all previous entries, K is assigned the value for the entry. The screen is CLEARed in statement 130. Statement 140 PRINTS the values for the Y axis. The

```
AT 1,1;K
```

in statement 140 PRINTs the highest bar value at the top left of the screen. The

```
AT 5,1;K*.75
```

in statement 140 PRINTs a number that is equal to three quarters of K, one quarter of the way from the top of the screen, etc.

Statements 150 through 280 form a FOR-NEXT loop that PRINTs the individual bars. Statements 160 through 250 are a FOR-NEXT loop nested in the first FOR-NEXT loop. The nested loop PRINTs a given bar in the graph. Statement 160 is used to print as many graphic symbols as necessary to display a given bar. Note that a bar starts at line 20 of the screen and successive symbols in the bar are placed in lines 19, 18, etc. The bar is finished when

```
(21-INT (A(I)*20/K))
```

is reached. Statements 170 through 220 are used to produce a different symbol for each of the six possible bars. Statement 230 prints the appropriate symbol for a bar starting in row 20 and column 7. Statement 240 decrements A so that the next symbol will be printed in the next higher row of the screen. After a bar is completed, statement 260 increases B by 4 so that the next bar will be to the

right of the preceding bar. Statement 270 restarts A at 20 so that the next bar can be started in line 20 of the screen. Statements 290 through 320 PRINT the numbers for the X-axis. Statement 290 starts printing at the seventh column of line 21. The FOR-NEXT loop in statements 300 through 320 PRINTs as many numbers as there are bars. Statement 310 PRINTs the number and makes space so that the next number will be printed under the next bar.

Chapter 17

Plotting

Plotting with AT

We have used AT to "plot" graphic symbols. For example

```
PRINT AT 4,3;"■"
```

PRINTs a black square at the fourth row and third column of the screen. After a graphic symbol is "plotted" we can "unplot" the symbol with a statement such as

```
PRINT AT 4,3;" "
```

AT permits "plotting" in a 22 by 32 grid. The grid elements for rows are labeled 0 through 21, and the grid elements for columns are labeled 0 through 31. Figure 17-1 shows the grid used for "plotting" with AT.

The black square in the grid could be "plotted" with

```
PRINT AT 19,29;"■"
```

The following program is for a little square that appears to inch across a row of the screen. After it reaches the end of the row it stops at the last position in the row. Then another square inches across the next row of the screen, etc. When the last square in the last row of the screen has inched across the screen, the program stops.

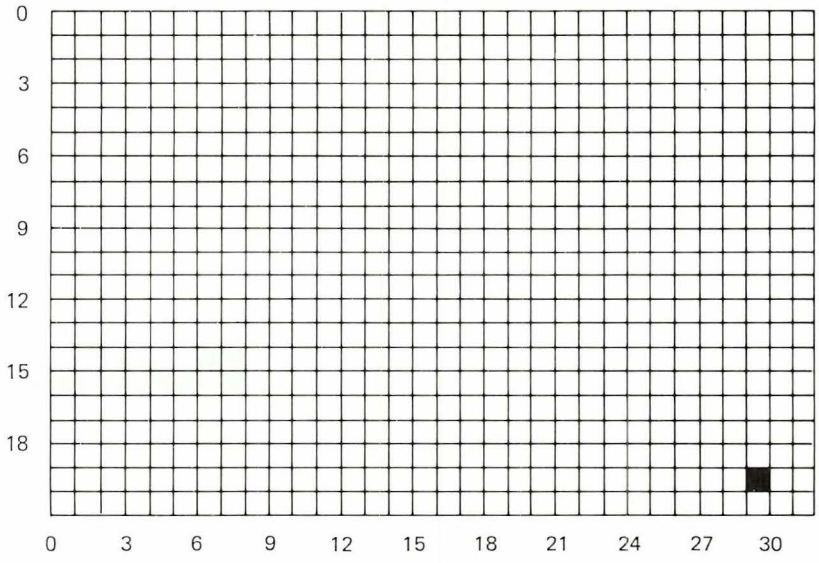


Figure 17-1. "Plotting" with AT

```
10  FOR J=0 TO 21
20  FOR K=0 TO 30
30  PRINT AT J,K;"■"
40  PRINT AT J,K+1;"■"
50  PRINT AT J,K;" "
60  NEXT K
70  NEXT J
80  STOP
```

If statement 80 is changed to

```
80  GOTO 10
```

the program will restart at the first line of the screen after a square has inched across the twenty-second row of the screen. If the following statement is added

```
65  CLS
```

the square for a given row will inch off the screen rather than stay in row 22. Change the program by switching statement 10 with

statement 20 and statement 60 with statement 70. The program becomes

```
10  FOR K=0 TO 30
20  FOR J=0 TO 21
30  PRINT AT J,K;"■"
40  PRINT AT J,K+1;"■"
50  PRINT AT J;K;" "
60  NEXT J
65  CLS
70  NEXT K
80  GOTO 10
```

Now the little square wobbles down the first column of the screen and leaves a trail. When the square reaches the bottom of the screen the trail is removed. A second square wobbles down the second column and leaves a trail. Remove statement 65 and columns subsequent to column 1 will appear to push the preceding column off the screen.

Change statements 10, 20, and 40 to

```
10  FOR K=0 TO 31
20  FOR J=0 TO 20
40  PRINT AT J+1,K;" "
```

Now the square inches down the columns in the same way that the square inched across the rows in the second "inching" program.

Picture Elements

Sinclair microcomputers have the capability of making graphs with greater detail than graphs that are produced with special graphics symbols. The PLOT keyword can be used to plot a little black space that is one quarter the size of a graphics symbol. The following figure shows the coordinates for picture elements.

Picture elements, like all other characters, are plotted in the top 22 rows of the screen. Since picture elements are half as large as graphic symbols, 44 picture elements can be fitted in the top 22 rows of the screen. Consider the COMMAND

```
PLOT 7,19
```

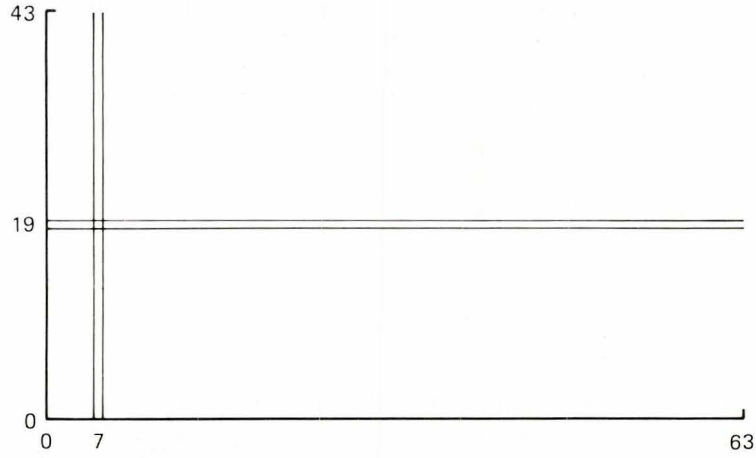


FIGURE 17-2. Coordinates for picture elements.

This COMMAND will plot a small black square at coordinates 7 and 19. The first coordinate is distance from 0 on the X axis, and the second coordinate is distance from 0 on the Y axis. The picture element given in the COMMAND is shown in Figure 17-2.

Now we can use the microcomputer to PLOT a SIN function. We know that SIN can take on values -1 through +1. The SIN function does not begin to repeat until a circle of 360 degrees is completed. Since the microcomputer works in radians, we shall substitute 2π for 360 degrees. Suppose we want to plot the sin function from 0 to 2.

We shall need a FOR-NEXT loop that goes from 0 through 63 because the X values can range from 0 to 63. Over every X value (0, 1, 2, 3, ... through 63) we shall PLOT the appropriate Y value. The Y value for X equal to 0 is the SIN of 0 radians. The Y value for X equal to 63 is 2π radians. Our completed program is

```
10  FOR X=1 TO 63
20  PLOT X, 22+20*SIN (X*2*PI/63)
30  NEXT X
40  STOP
```

Note that the FOR-NEXT loop will PLOT a Y value (function of X) above each of the 63 X values. Examine the formula for the Y value. The formula is

```
22+20+SIN (X*2*PI/63)
```

The expression

```
SIN (X*2*PI/63)
```

will have a value of 0 when X is 0 because

```
SIN (X*2*PI/63) = SIN (0*2*PI/63)
```

When X is 16

```
SIN (X*2*PI/63) = SIN (16*2*PI/63)
                = SIN 32* /63
```

Since 32/63 is approximately 32/64, one has

```
SIN (32*PI/63) = SIN PI/2
               = 1
```

Similarly SIN (X*2*PI/63) is 0 when X is 32, -1 when X is 48, and 0 when X is 63. Thus, SIN (X*2*PI/63) takes on values between -1 and +1. Since half of the Y values are negative, the 22 in

```
PLOT X, 22+20* SIN (X*2*PI/63)
```

raises the value of 0 so that it is plotted in the middle of the screen. Remember that 22 is half of 44, the number of positions on the Y axis. The number 20 expands the PLOT of Y values. When SIN (X*2*PI/63) is at a minimum or maximum (+1 or -1), 20 times this number is +20 or -20. If 10 is substituted for 20 in the formula, one will have a SIN function that only takes the middle half of the screen.

The program for PLOTting a SIN function can be changed to PLOT a COS function by substituting COS for SIN in the program.

UNPLOT

A PLOTted picture element can be removed from the screen with the UNPLOT keyword. Examine the following program

```
10  FOR X=1 TO 63
20  PLOT X,43* TAN (X*PI/4/63)
30  NEXT X
40  FOR X=1 TO 63
```

```
50  UNPLOT X,43* TAN (X*PI/4/63)
60  NEXT X
70  STOP
```

The program plots the TAN of angles from 0 radians (left of the screen) to PI/4 radians (right of the screen). Since all TANgents are for angles between 0 and PI/4, 0 on the Y axis can be at the bottom of the screen. The largest TANgent between 0 and PI/4 is 1. As a result, 43 is used as a multiplier so that the TAN of PI/4 will have a Y value that is at the top of the screen. After the TANgent is PLOTted, statements 40 through 60 UNPLOT the graph. Note that statements 40 through 60 are identical to statements 10 through 30, except that UNPLOT is substituted for PLOT. The UNPLOT keyword, like the PLOT keyword, automatically changes each Y value that is to be PLOTted into an integer. For example, if a Y value turns out to be 43.2, the microcomputer changes the Y value to 43. If a PRINT statement occurs after a PLOT or UNPLOT statement, the PRINT statement starts in the row of the PLOTted or UNPLOTted point, in the column immediately after the PLOTted or UNPLOTted point.

Straight Line Between Two Sets of Coordinates

The following program requests coordinates for two points and PLOTs a straight line between two points

```
10  PRINT "X COORDINATE FOR FIRST POINT?"
20  INPUT A
30  PRINT "Y COORDINATE FOR FIRST POINT?"
40  INPUT B
50  PRINT "X COORDINATE FOR SECOND POINT?"
60  INPUT C
70  PRINT "Y COORDINATE FOR SECOND POINT?"
80  INPUT D
90  CLS
100 LET F=-1
110 IF A<C THEN LET F =-F
120 IF A=C THEN GOTO 180
130 FOR X=A TO C STEP F
140 LET E=(D-B)/(C-A)
150 PLOT X, X*E+B-A*E
160 NEXT X
170 STOP
```

```
180  IF B<D THEN GOTO 220
190  LET F=D
200  LET D=B
210  LET B=F
220  FOR X=B TO D
230  PLOT A,B
240  LET B=B+1
250  NEXT X
260  STOP
```

The coordinates for the first point are A and B, and the coordinates for the second point are C and D. A straight line has the equation

$$y = sx + k$$

where s is (D-B)/(C-A), and k is B-A*E. One problem arises when the formula is used. If C-A is equal to 0 (that is, the line is perpendicular to the X axis), the division of (D-B) by (C-A) is impossible because C-A is 0. Statements 180 through 260 take care of this special case.

Now we can examine the program in detail. Statements 10 and 30 request the coordinates for the first point. These coordinates are entered in statement 20 and 40. Statements 50 and 70 request the coordinates for the second point. These coordinates are entered in statements 60 and 80. The screen is cleared in statement 90. Statement 100 sets F equal to -1. Statement 110 compares A to C. If A < C, F is changed to a positive 1. After statement 110 is executed F is negative 1 when C >= A and positive 1 when C < A. That is, if the X coordinate of the second point is less than the X coordinate of the first point, F is positive, otherwise F is negative.

Statement 120 checks for the special case described above and transfers control to statement 180 when this special case is in effect. Statements 120 through 160 PLOT the line in a FOR-NEXT loop. Note that the STEP in statement 130 can be positive (when C < A) or negative (when C > A).

The slope is calculated in statement 140 and the coordinates to be PLOTted are specified in statement 150, where X is the X coordinate, E is the slope, (D-B)/(C-A), and B-A*E is the Y intercept, k.

If the line is perpendicular to the X axis, statement 180 determines if the Y value for the first point is less than the Y value for the second point. If this condition is not met, the two Y values are switched. Statements 230 through 250 PLOT the line. A is always the first coordinate because the line is perpendicular to the X axis. The Y coordinates take on all values between B and D.

Circles

The following program generates a circle with a given radius and a given set of coordinates for the center.

```
10  PRINT "X COORDINATE OF CENTER?"
20  INPUT A
30  PRINT "Y COORDINATE OF CENTER?"
40  INPUT B
50  PRINT "RADIUS?"
60  INPUT C
70  CLS
80  FOR X=0 TO 96
90  PLOT A+C* SIN (X/48*PI),B+C* COS (X/48*PI)
100 NEXT X
110 STOP
```

Statement 80 is

```
80  FOR X=0 TO 96
```

This means that the program will PLOT a maximum of 96 points on the circle. Any even integer can be substituted for 96. However, the number in statement 90 has to be half of the number that is selected for statement 80. The larger the number after TO in statement 80, the more points on the circle. As this number becomes larger, the program takes longer to draw the circle. Note that we modify the Y and the X value for successive points on the circle. Variable C determines the radius, variable A determines the X coordinate of the center, and variable B determines the Y coordinate of the center.

Graphs With Graphic Symbols

One can "plot" graphs using the graphic symbols. However, this procedure is more difficult because the first value in the AT function is the number of rows from the top, not the bottom of the screen. Examine the following program for "plotting" the SINs of angles from 0 to 2π radians.

```
10  FOR X=0 TO 31
20  PRINT AT 10*(1-SIN (X*PI*2/31)),X;"■"
30  NEXT X
```

The AT statement requires the Y value (row) before the X value (column). SINs can take on numbers from -1 to +1. Since the AT statement counts from the top of the screen, all of the Y values for SIN must be subtracted from 1. The Y values for 1-SIN will still vary from +1 to -1. The 10 in statement 20 multiplies the Y value so that the Y value for SIN 1 is at the top of the screen, and the Y value for SIN -1 is at the bottom of the screen. Remember that there are only 22 rows (not 44) for an AT function. The graphic symbol for the AT function can be any of the graphics symbols, or any other character.

One could "plot" a COS function with the following program.

```
10  FOR X=0 TO 31
20  PRINT AT 10*(1-COS (X*PI*2/31)),X;"■"
30  NEXT X
```

One could "plot" half of a parabola with the following program.

```
10  FOR X=0 TO 31
20  PRINT AT 10*(2-SQR (X/8)),X;"■"
30  NEXT X
```

The number 2 is the maximum Y value that the parabola segment can attain.

Chapter 18

Slicing

Slicing Strings

Sinclair microcomputers make it possible to manipulate strings. Consider the following statement from a program.

```
20  LET A$="VARIANCE"
```

The string "VARIANCE" has 8 characters. If we add

```
30  LET B$=A$(2 TO 5)
```

to the program, B$ will become ARIA when the program is in RUN. That is, B$ is a new four character string that starts with the second character in A$ and ends with the fifth character in A$. If the user wants B$ to have all of the characters starting from the third character in A$ and continuing to the end of A$, she could change statement 30 to

```
30  LET B$=A$(3 TO 8)
```

or

```
30  LET B$=A$(3 TO)
```

The statement

```
30  LET B$=A$( TO 4)
```

is the same as

```
30  LET B$=A$(1 TO 4)
```

That is, the missing number before TO is assumed to be 1. The original string A$ can be replicated with

```
30  LET B$=A$( TO )
```

or

```
30  LET B$=A$()
```

or

```
30  LET B$=A$
```

A string consisting of one of the characters in another string can be obtained with a statement such as

```
30  LET B$=A$(4 TO 4)
```

If A$ is "VARIANCE", B$ becomes "I" when statement 30 is executed. The statement

```
30  LET B$=A$(4 TO 4)
```

can also be written as

```
30  LET B$=A$(4)
```

If the user specifies a starting number that is less than 1, an error report of 3 will occur. Thus

```
30  LET B$=A$(-5 TO 7)
```

is impossible. If the number after TO is less than the number before TO, the empty string will be produced. For example

```
30  LET B$=A$(7 TO 6)
```

will assign the empty string, "", to B$. If the number after TO is greater than the highest string position in the original string, an

error report of 3 will be generated. Thus, if A$ is "VARIANCE" and the statement

```
30  LET B$=A$(4 TO 19)
```

is executed, the computer will produce a report of 3. Part of a string can be modified. Consider the following sequence of statements.

```
10  LET Z$="THE SUM OF SQUARES"
20  LET Z$(5 TO 7)="PRD"
```

After statements 10 and 20 are executed, Z$ will be

```
"THE PRD OF SQUARES"
```

The sequence

```
10  LET Z$="THE SUM OF SQUARES"
20  LET Z$(5 TO 7)="PRODUCT"
```

will change Z$ to

```
"THE PRO OF SQUARES"
```

The computer cannot assign more than 3 characters to positions 5 through 7. Since it can only assign 3 characters, it assigns the first three characters in "PRODUCT".

Once a string is created, any additions to the string will produce a string of the same length as the original string. For example

```
10  LET A$="THE DEGREES OF FREEDOM"
20  LET A$(5 TO 22)=""
30  LET A$(5 TO 6)="MS"
```

will produce

```
A$="THE MS
```

after the program segment is RUN.

The notation involving a left parenthesis, a number or variable (or implied number or variable), TO, and a number or variable (or implied number or variable) is called slicing notation.

One can always reassign a string to A$. Thus

```
10  LET Z$="THE SUM OF SQUARES"
20  LET Z$(5 TO 7)="PRD"
30  LET Z$="THE MS"
```

will have "THE MS" assigned to Z$ after the above program sequence is RUN. The following sequence

```
 5  LET C$=" IS"
10  LET Z$="THE SUM OF SQUARES"
20  LET Z$=(5 TO 7)="PRD"
30  LET Z$="THE MS"
40  PRINT Z$+C$
```

will PRINT

```
THE MS IS
```

when the sequence is RUN.

Suppose that the sequence

```
10  LET Q$="THE MACHINE IS READY"
20  LET Q$(5 TO 11)="COMPUTER"
```

is entered. "MACHINE" occupies 7 positions (5 through 11) in the string in statement 10. "COMPUTER" is 8 letters long. When the above program sequence is RUN, the computer will discard the last letter in COMPUTER so that it can make a 7 position assignment to Q$ in statement 20. After statements 10 and 20 are executed, Q$ will contain

```
THE COMPUTE IS READY
```

In summary, if slicing notation is used after a string, the string retains its original length. If new characters are assigned to a string, the string will retain its original length. If the new characters exceed the number of characters that they are replacing, enough characters will be replaced (starting with the first new character) to keep the replacement characters the same length as the characters that they replace.

Slicing is high on the hierarchy of things that the computer does. A series of statements such as

```
10  LET A$="4+3*2"
20  LET Y=VAL A$(3 TO 5)
```

does not require that statement 20 be written as

```
20  LET Y=VAL (A$(3 TO 5))
```

When statements 10 and 20 are executed, Y will be assigned a value of 6.

Chapter 19

Strings in Arrays

String Arrays

Strings can be placed in arrays. The arrays can have any number of DIMensions. A DIM statement for an array requires that the length of each of the strings in the array is specified as the last DIMension. For example

```
10  DIM A$(3,2,7)
```

is a DIM statement for a two DIMensional array with 7 characters in each of the 6 array locations. Only one array can be DIMensioned in a given DIM statement.

A string array, like a string, is named with a single letter of the alphabet followed by a dollar sign. A string array name must not have the same name as any string in the program. Thus, A$ and A$(I) cannot be in the same program. Of course, the two DIMensional array Q$(I,J) cannot be in the same program as the four DIMensional array Q$(I,J,K,L).

Consider the array DIMensioned with

```
10  DIM C$(12,9)
```

It is a one DIMensional array with 9 spaces reserved for each of the 12 array locations. After the array is DIMensioned, no further reference need be made to the length of the array elements. The preceding DIMension statement will not allow an array element that is shorter or longer than 9 characters. The DIM statement must occur

earlier in the program than the first reference to an element or elements of the array.

Suppose that a program with DIM statement

```
5 DIM C$(7,6)
```

encounters

```
10  LET C$(4)="SUM"
```

The fourth of the 7 array elements will be assigned

```
"SUM  "
```

Since each array element is 6 characters long,

```
C$(4)="SUM   "
```

not

```
C$(4)="SUM"
```

Note that an ordinary string could be assigned

```
"SUM"
```

as opposed to

```
"SUM  "
```

Slicing String Arrays

Slicing notation can be used with string arrays. The notation

```
A$(4)(5 TO 7)
```

means the same thing as

```
A$(4,5 TO 7)
```

Both of the above examples of slicing refer to string element 4 in the same way as

Q$(5 TO 7)

refers to the string Q$.

Suppose that array element A$(1) contains "CARDS ". Also suppose that B$ contains an ordinary string, not a string array. One could obtain the fourth character in array element 1 with

```
LET B$=A$(1)(4)
```

or

```
LET B$=A$(1)(4 TO 4)
```

or

```
LET B$=A$(1,4)
```

or

```
LET B$=A$(1,4 TO 4)
```

All of the above examples would assign "D" to B$.

An array can be DIMensioned with no DIMensions. For example,

```
10  DIM R$(5)
```

will DIMension an "array" that is essentially a string that is 5 characters long.

Chapter 20

Recorded Programs

SAVING and LOADing Programs

A cassette tape recorder can be attached to the microcomputer by connecting the output jack of the recorder (labeled "ear" or "spk") to the "ear" jack on the computer. Then connect the "mic" jack on the recorder to the "mic' jack on the computer. If the recorder can be operated on battery or AC first try it with batteries. A recorder that will not work with the microcomputer while battery operated is most unlikely to work while AC operated. Some recorders require removing the "MIC" plug from the computer while loading a program from the cassette and/or removing the "ear" jack from the computer while saving (recording) a program to the cassette. If the recorder is a stereo recorder, record on only one track. Monaural recorders usually work better than stereo recorders. The microcomputer signal to the tape depends on two different amplitudes of sound recorded on the tape. For this reason, recorders that do not automatically adjust recording level are preferable to recorders that attempt to adjust recording level.

Use a high quality tape such as Memorex MRX1. Record as few programs as possible on a given tape. This will reduce loading time. If a tape is faulty, it will only interfere with one or two programs, rather than a series of programs. Record each program on the tape two or three times. If a tape is faulty, the second or third recording may be successfully entered in the computer.

Many recorders require that the recording level be set with precision. Buy a prerecorded tape from Sinclair. Adjust the volume control on the recorder so that the computer always loads this tape. Mark the volume control knob on the recorder so that it can always

be set in the proper position. After the recorder can be used to load a prerecorded tape, adjust the record level control (if your recorder has such a control) to record the tapes that can be played back at the level you have set on the volume control. This procedure will make it possible to record and play back your own tapes and commercial tapes without having to reset controls on the recorder.

The appropriate settings on the recorder controls must be determined empirically. Start with a control at the middle of its range and slowly increase (or decrease) the setting until good playback and good recordings are obtained. Set base controls to minimum and set treble controls to maximum. Inexpensive recorder usually work better than expensive recorders. They do not have automatic level adjustment, tone controls, and extra recording tracks for stereo recordings. However, some inexpensive recorders cannot maintain constant speed during recordings and/or introduce hiss on tape after it has been erased several times.

SAVing a Program

The following procedure is used to record a program on a cassette. Enter SAVE followed by the name of the program in quotation marks. For example

```
SAVE "VARIANCE"
```

Limit yourself to program names that meet the requirements of names for variables. Do not press ENTER until the recorder is placed in the record mode and the recorder is on long enough for the leader at the beginning of the tape to pass the record head. Then press ENTER. The recorder should respond with a 5 second period in which the screen is dark, followed by a period in which an inconsistent pattern, usually horizontal stripes, appears on the screen. After the program is SAVEd on cassette, the computer will produce a report of 0/0. The program should be recorded a second time. Leave some space on the tape before the second recording. Place a label on the cassette that includes the program name and the starting location for each program on the cassette. If the recorder has a counter that shows location on the cassette, be sure to zero this counter before the cassette is started. The number of the counter at the start of each recording can be written on the cassette next to the program name.

The program name can be recorded through the cassette microphone before the program is recorded. However, recording the

name requires disturbing the system by removing the plug in the microphone jack of the recorder. Such a procedure may produce more disadvantages than advantages. Repeatedly disconnecting the microphone plug can move the recorder enough to change the settings on its controls.

The first statement in a program can be made a REM statement that contains the name of the program. For example

```
10 REM "VARIANCE"
```

LOADING From a Cassette

Rewind the cassette to a location before the start of the required program. Then enter LOAD followed by the program name in quotation marks. For example

```
LOAD "VARIANCE"
```

Start the recorder by putting it in the play position and then depress the ENTER key.

If more than one program is recorded on the side of the cassette, the computer will ignore the programs with improper names and search until it finds the appropriate program. A program can also be loaded with

```
LOAD ""
```

followed by ENTER. Note that LOAD is followed by two sets of quotation marks, not the double quotation image used to print quotation marks inside of quotation marks. The above procedure will LOAD the first complete program that is encountered. However, the computer is less likely to LOAD a program correctly when it is not LOADed by name. Although a program can be loaded by placing two sets of quotation marks after LOAD, one cannot SAVE a program with

```
SAVE ""
```

The computer will return an error report of F when the name of the program that is being SAVEd is the empty string.

The user should note that the striped pattern on the screen during a LOAD is more consistent than the striped pattern during

a SAVE. If the screen does not look almost white during the five second period that precedes loading of a program, the playback level is set too high relative to the recording level taht was used for saving the program. During a LOAD one wants the 5 second lead period to produce a blank screen. The striped pattern that occurs during LOADing should produce the largest possible contrast between stripes and background.

Clearing Unnecessary Variables

When a program is SAVEd, all variables are SAVEd with the program. If the old values of the variables are not going to be used, enter the COMMAND

```
CLEAR
```

before SAVEing the program. CLEAR will remove the unnecessary variables and shorten what has to be saved on cassette. Since longer programs or programs with more variables take longer to LOAD, it is a good idea to CLEAR all variables when they are not needed. If variables are SAVEd with a program, do not RUN the program by entering

```
RUN
```

The RUN COMMAND will automatically CLEAR the variables that have been stored. The program can be RUN by entering GOTO followed by a statement number. If the user is not sure of the first statement number, she can enter

```
GOTO 1
```

RUNning the Program Automatically

The computer can load a program and automatically RUN the program after it is entered. This procedure will guarantee that program variables are not CLEARed when the program is RUN. Enter the program in the computer in the usual way. Terminate the program as in the following example

```
200 STOP
210 SAVE "VARIANCE"
220 GOTO 10
```

Note that the statement numbers for the three statements shown above can be any ascending statement numbers that are higher than the other statement numbers used in the program. Any program name can be used after the SAVE in statement 210. The number following GOTO in statement 220 is the number of the first statement in the program. The program is saved by entering the COMMAND

```
GOTO 210
```

That is, enter the COMMAND GOTO followed by the statement number of the statement that contains the keyword SAVE. Following the GOTO COMMAND the computer SAVEs the entire program. After the program is SAVEd control is returned to statement 220, the statement after the SAVE statement. This statement causes the computer to GOTO statement 10 to start executing the program. Since statement 220 is a GOTO statement rather than a RUN statement, variables are not cleared.

After the program is SAVEd it can be LOADed with the COMMAND

```
LOAD "VARIANCE"
```

The computer will LOAD the program and then execute the statement after the SAVE statement in the program. Since this statement is GOTO 10, the program starts execution at statement 10. The STOP in statement 200 STOPs the computer from executing the SAVE in the next statement.

Chapter 21

Saving Data

Entering Data in a Program

We have used the INPUT statement to enter data in a program. Suppose that a large number of observations have to be entered in a program. Part of the program could be

```
40  FOR I=1 TO 50
50  SCROLL
60  PRINT "OBSERVATION ";I;" IS ";
70  INPUT A(I)
80  PRINT I
90  NEXT I
```

The above sequence would enter the observations in array A. Of course, the program would require a DIM statement such as

```
10  DIM A(50)
```

The remainder of the program could use the numbers stored in array A. If the program is SAVEd, the numbers in array A are also SAVEd. When the program is LOADed from a cassette, it must be started with GOTO as opposed to RUN or the stored variables will be CLEARed. In the last chapter we discussed a way of automatically running a program without CLEARing variables. This approach can also be used to insure that array values are not lost because the RUN COMMAND is executed.

Before the program is SAVEd statements like 40 through 90 in the above example should be deleted. This will insure that the program will not request new data to replace the permanent data in the program.

Chapter 22

Controlling the Printer

Sinclair Printer

The LPRINT keyword is used just like the PRINT keyword. However, LPRINT causes PRINTing to occur at the printer rather than the television screen. If the user wants output to the printer and the screen, he can have two consecutive PRINT statements, one using PRINT and one using LPRINT.

COPY can be used in COMMAND or statement to COPY the current screen contents to the printer. LLIST can be used to produce a program listing on the printer. Similarly, LLIST followed by a statement number will start a printed listing at the statement number following LLIST. For example, if statement 75 exists

```
LLIST 75
```

will start the listing at statement 75. If statement 75 does not exist, the listing will start at the next higher program statement.

Index

BookWare™ *For the Timex Sinclair 1500, Timex Sinclair 1000 and ZX-81 Computers*

An excellent introduction to programming in BASIC on the Timex Sinclair 1500 and ZX-81. You'll learn about programming techniques, see examples that show you how BASIC commands work, and find out how to write your own programs. This easy-to-read book takes you step-by-step through the BASIC programming language. You'll be amazed at what you and the Timex Sinclair 1500 can do!

The series includes:

Software

Monarch!
Demolition/Ten-Pin
Championship Chess
CasinoPak I: One-Arm Bandit/ Blackjack
The Nowotnik Puzzle
Invasion Force
Escape from Shazzar!
Math Master: Self-Teaching Software for the Timex Sinclair 1500/1000

Books

The Timex Sinclair 1500/1000 Pocket Book
BASIC Basics of the Timex Sinclair 1500/1000
50 1K/2K Games for the Timex Sinclair 1500/1000
Basics of Timex Sinclair 1500/1000 BASIC
49 Explosive Games for the Timex Sinclair 1500/1000
Mastering Machine Code on Your Timex Sinclair 1500/1000
Making the Most of Your Timex Sinclair 1500/1000
Explorer's Guide to the Timex Sinclair 1500/1000
Timex Sinclair 1500/1000 Machine Language Programming and Interfacing
Exploring Timex Sinclair 1500/1000 Graphics

Reston Publishing Company, Inc.
A Prentice-Hall Company
Reston, Virginia

0-8359-0347-8

0 21898 03478 6